Contents

[**NLP** 4](#_Toc194939461)

[**Applications** 4](#_Toc194939462)

[**Regex for NLP** 5](#_Toc194939463)

[**Examples of Regex Usage in NLP:** 5](#_Toc194939464)

[**Key Concepts in Regex:** 5](#_Toc194939465)

[**NLP Core Concepts** 7](#_Toc194939466)

[**Transformer Models** 14](#_Toc194939467)

[**Self-Attention Mechanism** 14](#_Toc194939468)

[**Retrieval-Augmented Generation (RAG)** 16](#_Toc194939469)

[**RAG Summary** 18](#_Toc194939470)

[**Sentence Embedding** 19](#_Toc194939471)

[**Word Embedding** 20](#_Toc194939472)

[**NLP Pipeline** 21](#_Toc194939473)

[**Sentiment Analysis** 22](#_Toc194939474)

[**Vocabulary & Feature Extraction** 23](#_Toc194939475)

[**Feature Extraction with Frequencies** 23](#_Toc194939476)

[**Preprocessing** 25](#_Toc194939477)

[**Tokenization** 26](#_Toc194939478)

[**Stemming** 26](#_Toc194939479)

[**Types of Stemming Algorithms** 26](#_Toc194939480)

[**1. Porter Stemmer** 26](#_Toc194939481)

[**2. Lancaster Stemmer** 26](#_Toc194939482)

[**3. Snowball Stemmer (aka Porter2)** 26](#_Toc194939483)

[**4. Regex-based Stemmers** 27](#_Toc194939484)

[**Lemmatization** 27](#_Toc194939485)

[**NOTE:** 27](#_Toc194939486)

[**Parts of Speech** 28](#_Toc194939487)

[**1. POS Categories** 28](#_Toc194939488)

[**2. POS Tagging Approaches** 29](#_Toc194939489)

[**A. Rule-Based POS Tagging** 29](#_Toc194939490)

[**B. Statistical POS Tagging** 29](#_Toc194939491)

[**C. Machine Learning-Based POS Tagging** 29](#_Toc194939492)

[**3. POS Tagging in Python** 30](#_Toc194939493)

[**4. Advanced POS Tagging Techniques** 31](#_Toc194939494)

[**5. Why POS Tagging Matters?** 32](#_Toc194939495)

[**6. Common Challenges in POS Tagging** 32](#_Toc194939496)

[**Final Wisdom** 33](#_Toc194939497)

[**Named Entity Recognition (NER)** 34](#_Toc194939498)

[**1. Common Named Entity Categories** 34](#_Toc194939499)

[**2. Approaches to NER** 34](#_Toc194939500)

[**A. Rule-Based NER** 34](#_Toc194939501)

[**B. Statistical & Machine Learning-Based NER** 35](#_Toc194939502)

[**C. Deep Learning-Based NER** 35](#_Toc194939503)

[**3. NER Implementation in Python** 35](#_Toc194939504)

[**A. Using spaCy (Fast & Efficient)** 35](#_Toc194939505)

[**B. Using Hugging Face Transformers (State-of-the-Art)** 36](#_Toc194939506)

[**C. Using Stanford NER (CRF-Based)** 36](#_Toc194939507)

[**4. Advanced NER Techniques** 37](#_Toc194939508)

[**A. BiLSTM + CRF (Deep Learning + Structured Output)** 37](#_Toc194939509)

[**B. Transformer-Based NER (State-of-the-Art)** 37](#_Toc194939510)

[**5. Challenges in NER** 38](#_Toc194939511)

[**6. When to Use Which NER Model?** 38](#_Toc194939512)

[**7. Applications of NER** 39](#_Toc194939513)

[**Final Wisdom** 39](#_Toc194939514)

[**Text Representation (Feature Engineering)** 40](#_Toc194939515)

[**1. Why Do We Need Text Representation?** 40](#_Toc194939516)

[**2. Types of Text Representation** 40](#_Toc194939517)

[**A. Basic Text Representations (Classical Methods)** 40](#_Toc194939518)

[**B. Dense Word Embeddings (Advanced Methods)** 42](#_Toc194939519)

[**C. Contextual Word Embeddings (State-of-the-Art)** 43](#_Toc194939520)

[3. Comparing Text Representation Methods 44](#_Toc194939521)

[**4. Choosing the Right Text Representation** 44](#_Toc194939522)

[**Final Wisdom** 45](#_Toc194939523)

[**Bayes Rule** 46](#_Toc194939524)

[**Log Likelihood** 48](#_Toc194939525)

[**Naïve Bayes Assumptions** 49](#_Toc194939526)

[**Error Analysis** 50](#_Toc194939527)

[**TF-IDF** 52](#_Toc194939528)

# **NLP**

NLP, or Natural Language Processing, is a field of artificial intelligence that focuses on enabling computers to understand, interpret, and generate human language, allowing for interaction and tasks like translation and text analysis.

## **Applications**

* **Language Translation:** Translating text or speech from one language to another.
* **Sentiment Analysis:** Determining the emotional tone or opinion expressed in a piece of text.
* **Question Answering:** Allowing computers to understand and answer questions posed in natural language.
* **Speech Recognition:** Converting spoken words into text.
* **Chatbots and Virtual Assistants:** Enabling computers to engage in natural language conversations.
* **Text Summarization:** Condensing long texts into shorter, more manageable summaries.
* **Named Entity Recognition:** Identifying and classifying entities like people, organizations, and locations in text.
* **Autocompletion**
* **Spam Filtration**

# **Regex for NLP**

Regular expressions (regex) are powerful tools in Natural Language Processing (NLP) for tasks like text cleaning, pattern matching, data extraction, and validation, enabling efficient manipulation and analysis of text data.

* A regular expression (regex) is a sequence of characters that defines a search pattern.
* They are used to find, extract, and manipulate text based on specific patterns.
* Think of them as a language for specifying text search strings.

They are used for Text Preprocessing, Pattern Matching, Data Extraction, Data Validation, and Information Retrieval.

## **Examples of Regex Usage in NLP:**

* **Extracting email addresses:** [\w\.-]+@[\w\.-]+\.\w+
* **Finding phone numbers:** \d{3}-\d{3}-\d{4} or \(\d{3}\) \d{3}-\d{4}
* **Removing punctuation:** [^a-zA-Z0-9\s]
* **Tokenizing text:** Splitting text into words or sentences based on spaces or punctuation.
* **Identifying specific words:** \bcat\b (finds the word "cat" as a whole word, not part of another word like "cats")

## **Key Concepts in Regex:**

* **Metacharacters:**

Special characters with specific meanings, like **.** (any character), **\*** (zero or more occurrences), **+** (one or more occurrences), **?** (zero or one occurrence), **[]** (character class), **()** (grouping).

* **Character Classes:**

Sets of characters, like **[a-z]** (lowercase letters), **[0-9]** (digits), **\s** (whitespace).

* **Quantifiers:**

Symbols that specify how many times a character or group should appear, like **\*, +, ?, {n}, {n,}, {n,m}**.

* **Anchors:**

Characters that specify the position of a match, like **^** (beginning of string), **$** (end of string), **\b** (word boundary).

# **NLP Core Concepts**

**1. Tokenization**

Tokenization is the process of breaking text into smaller units (tokens), which can be **words, subwords, or sentences**.

**Types of Tokenization**

1. **Word Tokenization**
   * Splitting a sentence into words.
   * Example:
   * Text: "I love NLP!"
   * Word Tokens: ['I', 'love', 'NLP', '!']
2. **Subword Tokenization**
   * Splitting words into meaningful subwords (used in BERT, GPT).
   * **Example:**
   * Word: "unhappiness"
   * Subword Tokens: ['un', 'happiness']
3. **Sentence Tokenization**
   * Splitting text into sentences.
   * **Example:**
   * Text: "I love NLP. It's amazing!"
   * Sentence Tokens: ["I love NLP.", "It's amazing!"]

📌 **Libraries for Tokenization**:

* nltk.word\_tokenize(text), nltk.sent\_tokenize(text)
* spacy tokenizer
* Hugging Face tokenizers for subword models

**2. Stopwords Removal**

Stopwords are common words (e.g., *the, is, in, at, which, and*) that don’t add much meaning and can be removed to improve efficiency.

**Example**

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

text = "This is a simple NLP example"

tokens = word\_tokenize(text)

filtered\_tokens = [word for word in tokens if word.lower() not in stopwords.words('english')]

print(filtered\_tokens) # ['simple', 'NLP', 'example']

📌 **Stopwords Lists**:

* nltk.corpus.stopwords.words('english')
* spacy has built-in stopwords

**3. Stemming vs Lemmatization**

Both techniques **reduce words to their base/root form**, but they work differently.

**Stemming**

* Removes suffixes but may not return valid words.
* Uses **heuristic rules**.
* Example (Porter Stemmer):
* from nltk.stem import PorterStemmer
* stemmer = PorterStemmer()
* print(stemmer.stem("running")) # Output: run
* print(stemmer.stem("flies")) # Output: flies

**Lemmatization**

* Uses a **dictionary lookup** to return valid words.
* More accurate than stemming.
* Example (WordNet Lemmatizer):
* from nltk.stem import WordNetLemmatizer
* lemmatizer = WordNetLemmatizer()
* print(lemmatizer.lemmatize("running", pos="v")) # Output: run
* print(lemmatizer.lemmatize("flies", pos="n")) # Output: fly

📌 **When to Use?**

* **Stemming**: If speed matters and accuracy isn’t critical.
* **Lemmatization**: If you need **correct words** (better for NLP tasks like search engines).

**4. POS Tagging (Part of Speech Tagging)**

Assigns **grammatical labels** (noun, verb, adjective, etc.) to words in a sentence.

**Example using spaCy**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "The quick brown fox jumps over the lazy dog."

doc = nlp(text)

for token in doc:

print(token.text, "->", token.pos\_)

**Output:**

The -> DET

quick -> ADJ

brown -> ADJ

fox -> NOUN

jumps -> VERB

over -> ADP

the -> DET

lazy -> ADJ

dog -> NOUN

📌 **Why is POS Tagging Useful?**

* Improves Named Entity Recognition (NER)
* Helps in **lemmatization** (knowing "running" is a verb helps get "run")
* Used in **syntactic parsing**

**5. Named Entity Recognition (NER)**

NER identifies and categorizes entities (names, places, dates, organizations, etc.).

**Example using spaCy**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "Elon Musk founded SpaceX in 2002."

doc = nlp(text)

for ent in doc.ents:

print(ent.text, "->", ent.label\_)

**Output:**

Elon Musk -> PERSON

SpaceX -> ORG

2002 -> DATE

📌 **NER Applications**:

* Extracting **company names** from resumes
* **Medical NLP** (finding diseases, drugs in medical text)
* Chatbot intelligence (identifying **locations, names** in user input)

**6. Dependency Parsing**

* Analyzes the **grammatical structure** of a sentence.
* Determines how words relate to each other.

**Example using spaCy**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "The cat sat on the mat."

doc = nlp(text)

for token in doc:

print(token.text, "->", token.dep\_, "->", token.head.text)

**Output:**

The -> det -> cat

cat -> nsubj -> sat

sat -> ROOT -> sat

on -> prep -> sat

the -> det -> mat

mat -> pobj -> on

📌 **Key Terms**

* **ROOT**: The main verb of the sentence
* **nsubj**: Nominal subject (who/what performs the action)
* **det**: Determiner (e.g., *the, a*)
* **pobj**: Object of the preposition

✅ **Applications**

* **Question answering systems**
* **Machine translation**
* **Grammar checking tools**

**7. TF-IDF & Bag of Words (BoW)**

**Bag of Words (BoW)**

* Converts text into a **vector of word counts**.
* Ignores order but captures frequency.

**Example**

| **Sentence** | **NLP** | **is** | **fun** |
| --- | --- | --- | --- |
| "NLP is fun" | 1 | 1 | 1 |
| "NLP is great" | 1 | 1 | 0 |

📌 **Limitations**

* Doesn’t consider **word importance**.
* Large vocab size → **high-dimensional vectors**.

**TF-IDF (Term Frequency-Inverse Document Frequency)**

TF-IDF improves BoW by giving **more weight to rare words** and **less weight to common words**.

* **TF (Term Frequency)** = (Word count in doc) / (Total words in doc)
* **IDF (Inverse Document Frequency)** = log (Total docs / Docs containing word)

Example:

* "NLP" appears in 5 out of 10 documents → Low IDF score.
* "Transformer" appears in 1 out of 10 documents → High IDF score.

**Example using Sklearn**

from sklearn.feature\_extraction.text import TfidfVectorizer

docs = ["NLP is fun", "NLP is great"]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(docs)

print(vectorizer.get\_feature\_names\_out()) # ['fun', 'great', 'is', 'nlp']

print(tfidf\_matrix.toarray()) # TF-IDF values

✅ **TF-IDF vs BoW?**

* **BoW** is simpler and used in **basic NLP tasks**.
* **TF-IDF** is better when **word importance** matters (e.g., search engines).

# **Transformer Models**

Transformers revolutionized NLP by replacing RNNs and LSTMs. They use the **self-attention mechanism** and **positional encoding** to process text in parallel, leading to models like **BERT, GPT, and T5**.

🔹 **Key Components of a Transformer**

1. **Self-Attention Mechanism** – Enables the model to focus on important words.
2. **Multi-Head Attention** – Enhances attention by using multiple perspectives.
3. **Positional Encoding** – Adds order information to words.
4. **Feedforward Layers** – Used after self-attention for transformation.
5. **Layer Normalization & Dropout** – Improves training stability.

## **Self-Attention Mechanism**

Self-attention allows a model to weigh the importance of each word relative to others in a sentence. This is key for understanding context.

**How Does It Work?**

Given a sentence:  
💬 *"The cat sat on the mat."*

* Traditional models might struggle to capture dependencies like **"The cat"**.
* **Self-attention** enables the model to learn which words are important for each token.

**Mathematical Formulation**

For each input word (token), we compute:

1. **Query (QQ), Key (KK), and Value (VV) Matrices**
   * These are learned transformations of input embeddings.
   * If input dimension = dmodeld\_{model}, then:
     + Q=XWQQ = XW\_Q
     + K=XWKK = XW\_K
     + V=XWVV = XW\_V  
       (where WQ,WK,WVW\_Q, W\_K, W\_V are weight matrices)
2. **Attention Score Calculation (Scaled Dot-Product Attention)**
   * Compute scores using dot product of Query and Key: Attention(Q,K,V)=softmax(QKTdk)V\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d\_k}} \right) V
   * dkd\_k is the dimension of keys (used for scaling).
   * **Softmax** ensures attention weights sum to 1.

**Example**

If we process the sentence *"The cat sat on the mat."*

* The word **"cat"** attends to **"sat"** with high weight.
* **"The"** attends more to **"cat"**, but not much to **"on the mat"**.

✅ **Benefits of Self-Attention**

* Captures **long-range dependencies** in sentences.
* Works in **parallel**, unlike RNNs (which process sequentially).
* Used in **BERT (bi-directional context)** and **GPT (auto-regressive context)**.

# **Retrieval-Augmented Generation (RAG)**

RAG enhances text generation by **retrieving relevant information** before generating a response.

**Why RAG?**

* LLMs (e.g., GPT) have **fixed knowledge** (limited to training data).
* RAG **fetches external knowledge** dynamically, improving accuracy.
* Useful in **chatbots, search engines, question answering, summarization**.

**How Does RAG Work?**

RAG has **two main components**:

1. **Retriever**
   * Uses a **vector database** (like FAISS, Pinecone, ChromaDB)
   * Searches for relevant documents based on user input
   * Example: A search engine finds the top 5 Wikipedia pages for a query.
2. **Generator**
   * Uses an **LLM (GPT, T5, etc.)**
   * Combines retrieved knowledge with the prompt
   * Generates a final answer

**Example Workflow**

1. User asks: *"Who is the CEO of OpenAI?"*
2. **Retriever** finds recent articles mentioning OpenAI's CEO.
3. **Generator** synthesizes a response using retrieved data.

**Mathematical View of RAG**

P(y∣x,z)=P(y∣x,R(x))P(y | x, z) = P(y | x, R(x))

Where:

* xx = user query
* R(x)R(x) = retrieved documents
* yy = final generated response

📌 **Vector Search in RAG**

* Uses **word embeddings** to compare text similarity.
* Converts text to vectors using **BERT embeddings**.
* Stores & retrieves from **vector databases (FAISS, Pinecone, Weaviate)**.

**RAG Implementation Example**

from transformers import RagTokenizer, RagRetriever, RagSequenceForGeneration

# Load model

tokenizer = RagTokenizer.from\_pretrained("facebook/rag-sequence-nq")

retriever = RagRetriever.from\_pretrained("facebook/rag-sequence-nq")

model = RagSequenceForGeneration.from\_pretrained("facebook/rag-sequence-nq", retriever=retriever)

# Input query

query = "What is the capital of France?"

inputs = tokenizer(query, return\_tensors="pt")

# Generate response

generated = model.generate(\*\*inputs)

response = tokenizer.batch\_decode(generated, skip\_special\_tokens=True)

print(response) # Output: "The capital of France is Paris."

## **RAG Summary**

|  |  |
| --- | --- |
| **Concept** | **Explanation** |
| **Self-Attention** | Helps a model focus on important words in a sequence. |
| **Multi-Head Attention** | Uses multiple attention heads for better representation. |
| **RAG (Retrieval-Augmented Generation)** | Combines search (retrieval) with LLM generation. |
| **Vector Databases** | Stores and retrieves embeddings for similarity search. |

# **Sentence Embedding**

Sentence embedding is a technique in Natural Language Processing (NLP) that represents a sentence as a numerical vector, capturing its meaning and context, which can be used for tasks like semantic similarity analysis and text classification.

**What it is:**

Sentence embedding transforms a sentence into a fixed-length vector of real numbers, where each number represents a specific aspect of the sentence's meaning.

**How it works:**

Sentence embeddings are typically created using pre-trained models, like the Universal Sentence Encoder (USE) from Google, which have been trained on large amounts of text data.

**Example:**

Imagine you have two sentences: "The cat sat on the mat" and "A feline was seated on a rug". A sentence embedding model would likely produce similar vectors for these sentences because they convey similar meanings, even though the words are different.

**Applications:**

* **Search Engines:** To find documents that are semantically similar to a user's query.
* **Question Answering Systems:** To understand the meaning of a question and find relevant answers.
* **Chatbots:** To understand the user's intent and respond appropriately.
* **Text Summarization:** To identify the most important sentences in a document.

# **Word Embedding**

Word embedding is a technique in natural language processing (NLP) that represents words as numerical vectors (or "embeddings") in a multi-dimensional space, where similar words are positioned closer together, capturing semantic relationships.

**What it is:**

Word embeddings are a way to map words to vectors, allowing computers to understand the meaning and relationships between words.

**Why it's useful:**

* **Semantic Understanding:** Words with similar meanings (e.g., "king" and "queen") will have vectors that are closer together in the vector space.
* **Machine Learning:** NLP models can use these numerical representations as input, enabling them to perform tasks like sentiment analysis, text classification, and machine translation.

**How it works:**

* **Vector Space:** Each word is assigned a vector (a list of numbers) representing its meaning.
* **Distance and Similarity:** The distance between vectors reflects the similarity between words.
* **Training:** Word embedding models are trained on large amounts of text data to learn these vector representations.

**Examples:**

* **Word2Vec:** A popular algorithm for creating word embeddings, using techniques like Continuous Bag of Words (CBOW) and Skip-gram.
* **GloVe:** Another algorithm that uses global word co-occurrence statistics to learn word embeddings.

**Applications:**

* Sentiment Analysis
* Text Classification
* Machine Translation
* Information Retrieval

# **NLP Pipeline**

An NLP (Natural Language Processing) pipeline is a series of interconnected steps that transform raw text data into a format suitable for analysis or application, much like an assembly line. It typically involves steps like text preprocessing, feature extraction, and modeling.

Here’s a breakdown of the common stages in an NLP pipeline:

1. Data Acquisition
2. Text Preprocessing
3. Feature Engineering
4. Modelling
5. Evaluation
6. Deployment

**Points to remember:**

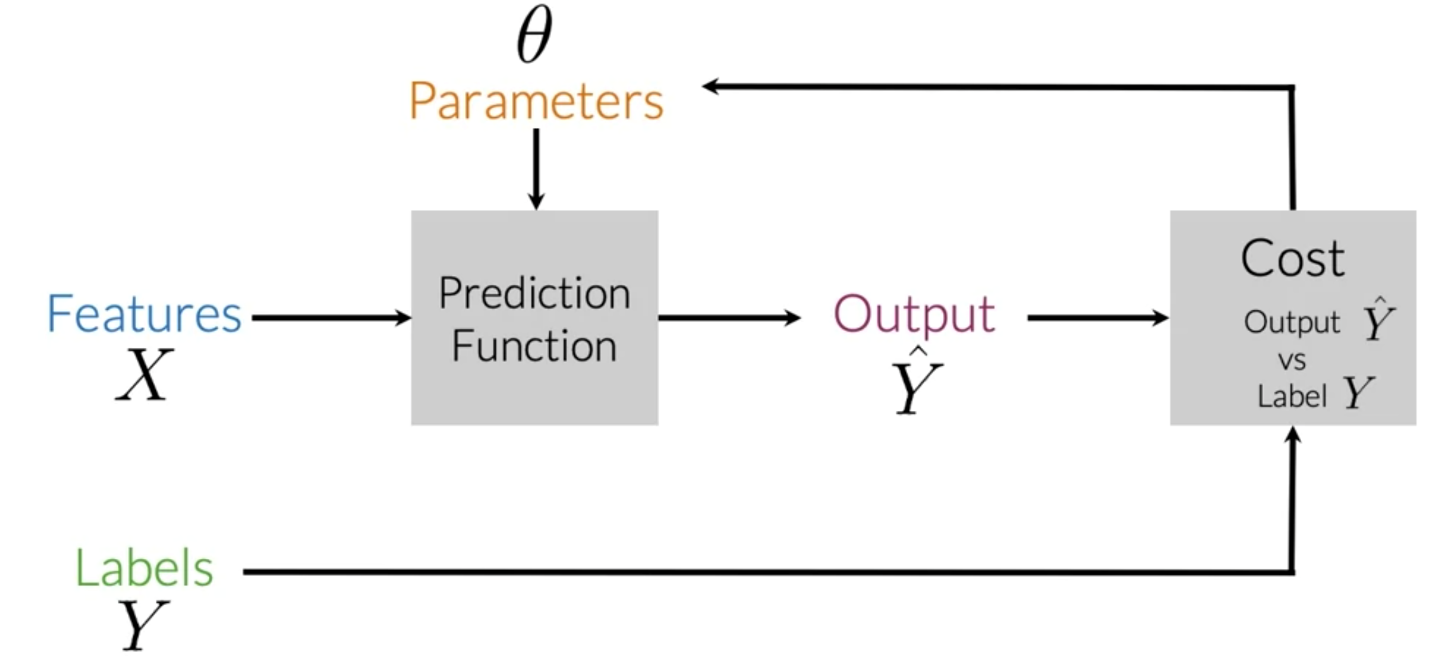
* This pipeline is not universal.
* This is ML pipeline and deep learning pipelines are slightly different.
* NLP pipeline is non-linear (that means stages can have more dynamic connections, allowing for branching and iteration).

# **Sentiment Analysis**

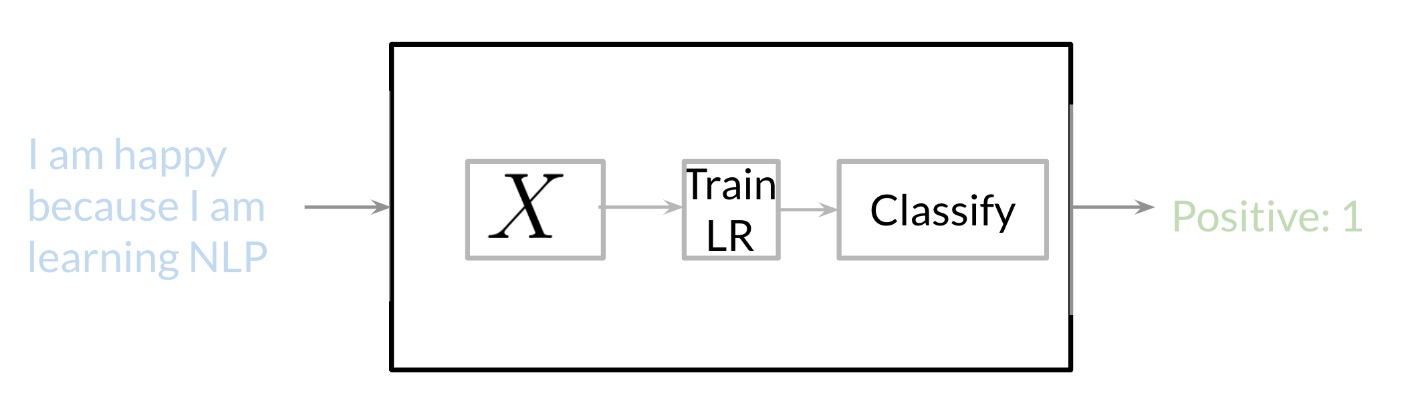
Sentiment analysis using machine learning (ML) involves automatically analyzing text, voice, or video to determine the emotional tone (positive, negative, or neutral) expressed, leveraging natural language processing (NLP) and ML techniques.

To determine the sentiment or emotional tone expressed in text, voice, or video.

In supervised machine learning, you usually have an input X, which goes into your prediction function to get your Y^. You can then compare your prediction with the true value Y. This gives you your cost which you use to update the parameters *θ*. The following image, summarizes the process.

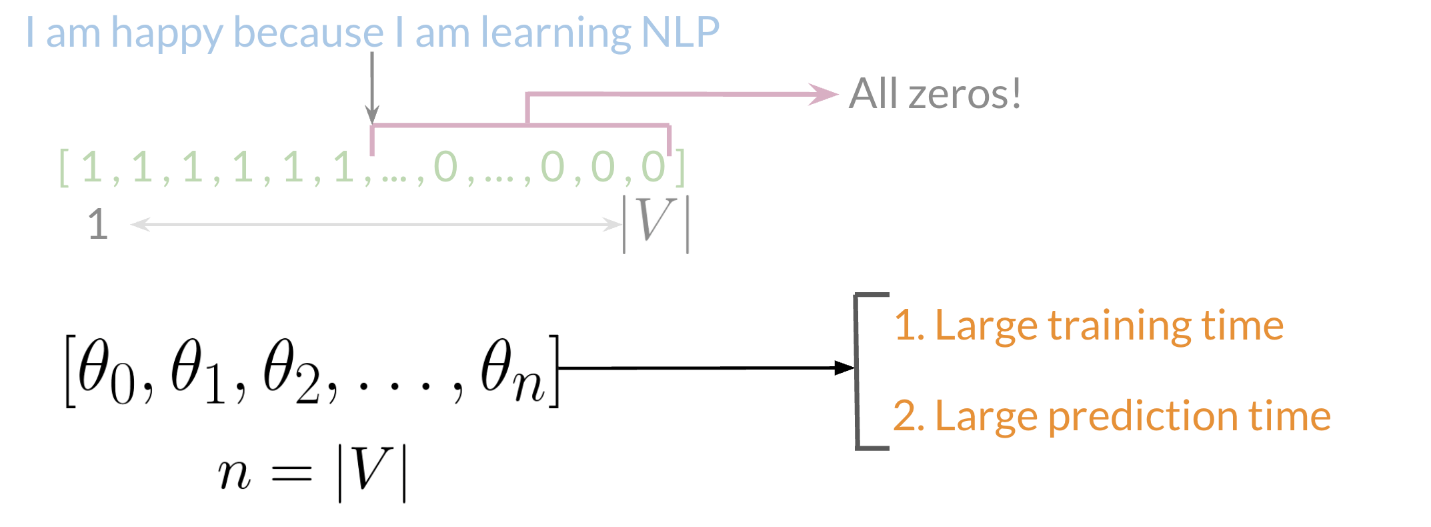


To perform sentiment analysis on a tweet, you first have to represent the text (i.e. "I am happy because I am learning NLP ") as features, you then train your logistic regression classifier, and then you can use it to classify the text.



# **Vocabulary & Feature Extraction**

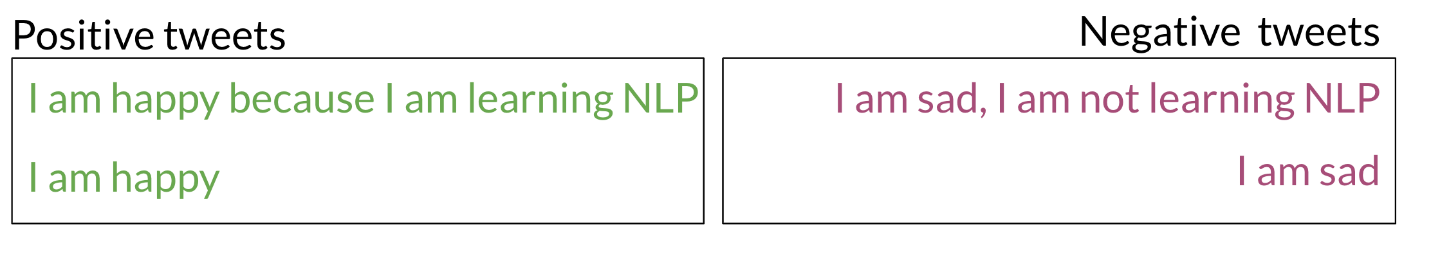
Given a tweet, or some text, you can represent it as a vector of dimension V, where V corresponds to your vocabulary size. If you had the tweet "I am happy because I am learning NLP", then you would put a 1 in the corresponding index for any word in the tweet, and a 0 otherwise.



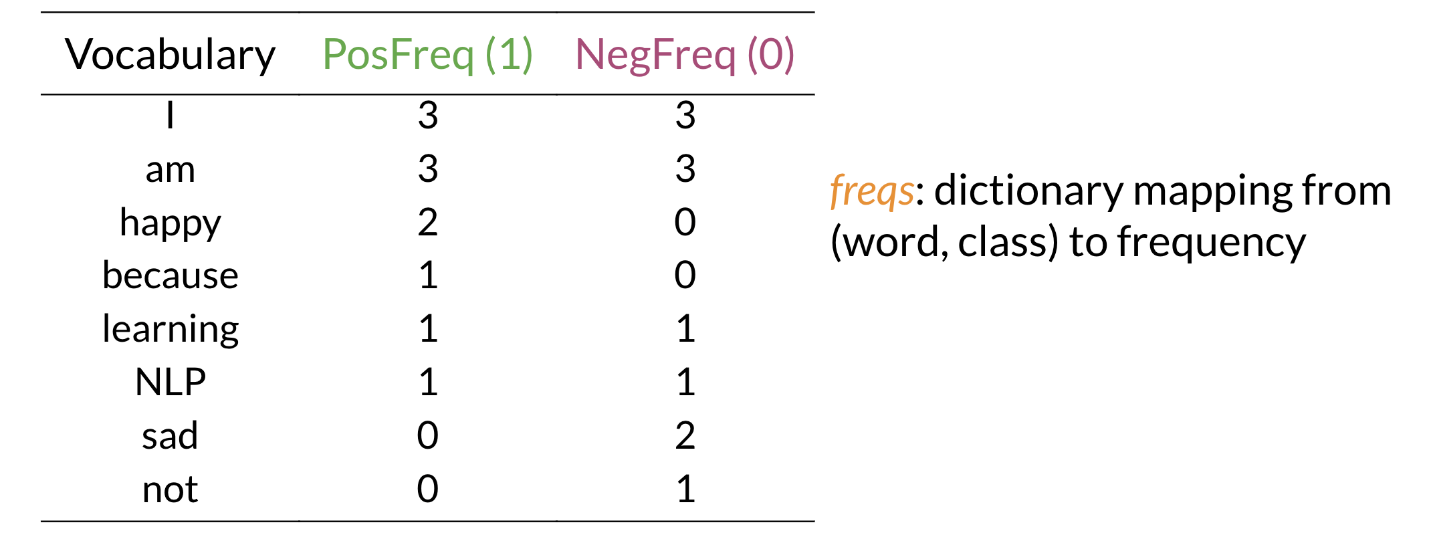
As you can see, as V gets larger, the vector becomes more sparse. Furthermore, we end up having many more features and end up training θ V parameters. This could result in larger training time, and large prediction time.

## **Feature Extraction with Frequencies**

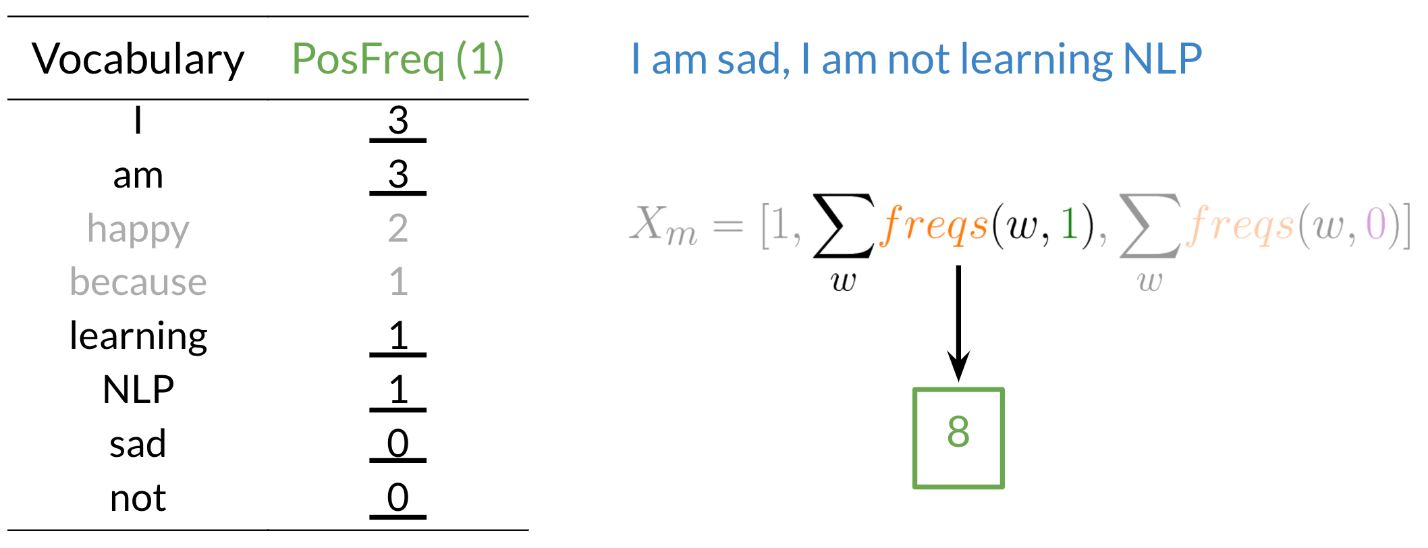
Given a corpus with positive and negative tweets as follows:



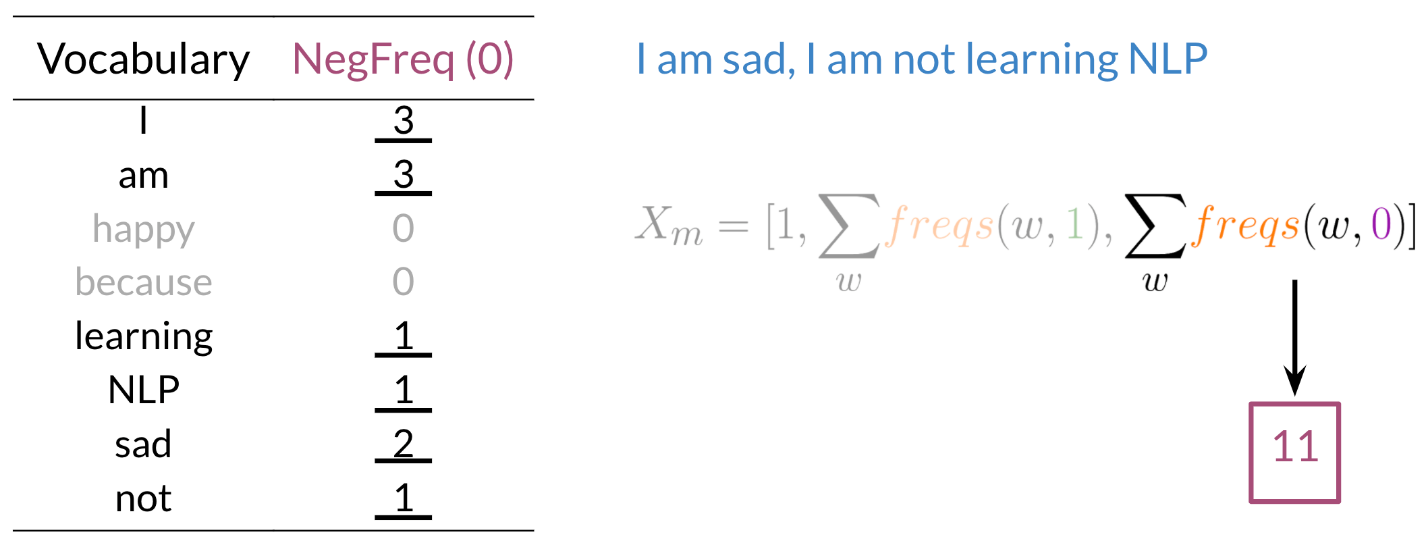
You have to encode each tweet as a vector. Previously, this vector was of dimension V. Now, as you will see in the upcoming videos, you will represent it with a vector of dimension 3. To do so, you have to create a dictionary to map the word, and the class it appeared in (positive or negative) to the number of times that word appeared in its corresponding class.



In the table above, you can see how words like happy and sad tend to take clear sides, while other words like "I, am" tend to be more neutral. Given this dictionary and the tweet, "I am sad, I am not learning NLP", you can create a vector corresponding to the feature as follows:



To encode the negative feature, you can do the same thing.



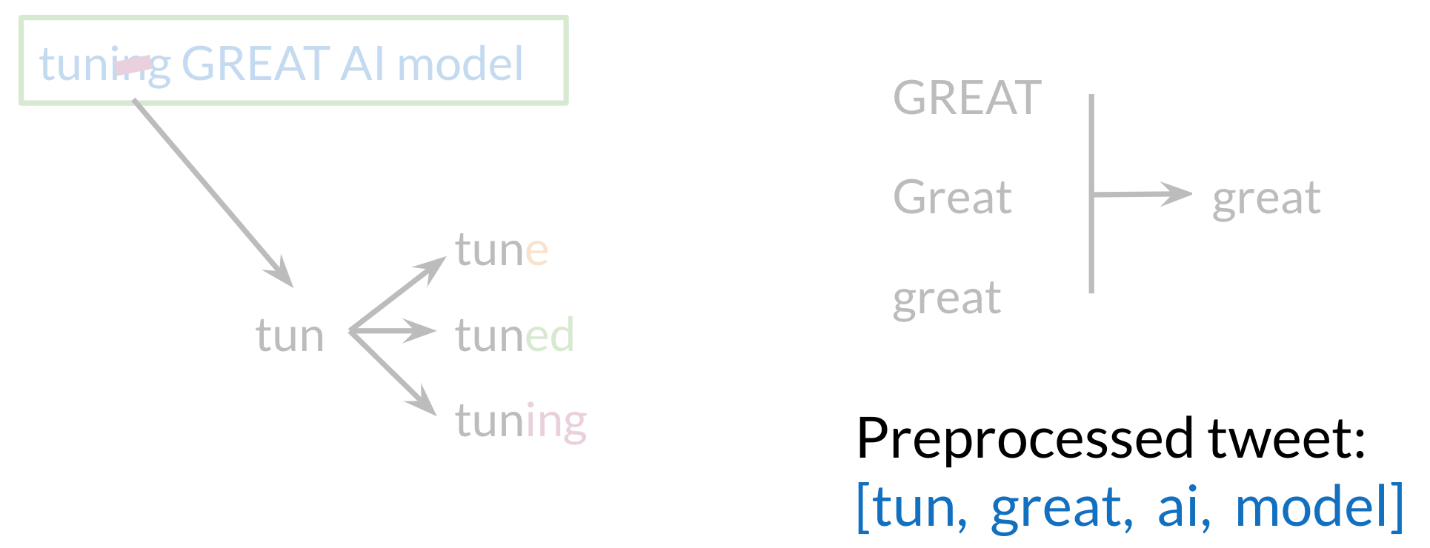
Hence you end up getting the following feature vector [1,8,11]. 1 corresponds to the bias, 8 the positive feature, and 11 the negative feature.

# **Preprocessing**

When preprocessing, you have to perform the following:

1. Eliminate handles and URLs
2. Tokenize the string into words.
3. Remove stop words like "and, is, a, on, etc."
4. Stemming - or convert every word to its stem. Like dancer, dancing, danced, becomes 'danc'. You can use porter stemmer to take care of this.
5. Convert all your words to lower case.

For example the following tweet "@YMourri and @AndrewYNg are tuning a GREAT AI model at https://deeplearning.ai!!!" after preprocessing becomes:



[tun, great, ai, model]. Hence you can see how we eliminated handles, tokenized it into words, removed stop words, performed stemming, and converted everything to lower case.

# **Tokenization**

It is a process of splitting text into meaningful segments.

**Sentence Tokenization** 🡪 Paragraph into sentences.

**Word Tokenization** 🡪 Sentences into words.

# **Stemming**

Stemming chops off affixes (prefixes or suffixes) to approximate the root form of a word. It is a rule-based, heuristic process that does not always produce valid words.

## **Types of Stemming Algorithms**

### **1. Porter Stemmer**

Uses a series of suffix-stripping rules. Works well but may over-stem (cut too much).

* Fast and computationally cheap.
* Works well for search engines where precision is not critical.

**Example:**

"running" → "run"

"better" → "better" (incorrectly retained)

"organization" → "organ" (over-stemming)

### **2. Lancaster Stemmer**

More aggressive than Porter.

Example:

"running" → "run"

"organization" → "org"

### **3. Snowball Stemmer (aka Porter2)**

Improved and more flexible version of the Porter Stemmer.

### **4. Regex-based Stemmers**

Custom rule-based stemming using regex patterns.

# **Lemmatization**

Lemmatization reduces words to their dictionary form (lemma) based on morphological analysis. Unlike stemming, it considers part of speech (POS) and meaning.

* Uses a vocabulary and morphological analysis.
* Requires POS tagging to work correctly.
* More accurate than stemming.
* Produces real words.
* Context-aware.

These two libraries are used for stemming **WordNet Lemmatizer (NLTK) and SpaCy Lemmatizer.**

**Example:**

"running" → "run"

"better" → "good"

"geese" → "goose"

"corpora" → "corpus"

### **NOTE:**

Use **stemming** for **speed** when approximate matching is fine (e.g., search engines).

Use **lemmatization** for accurate NLP tasks like sentiment analysis, text summarization, and named entity recognition.

# **Parts of Speech**

POS tagging assigns **grammatical categories** (nouns, verbs, adjectives, etc.) to words in a sentence. It's crucial for syntactic parsing, named entity recognition, machine translation, and more.

## **1. POS Categories**

|  |  |  |
| --- | --- | --- |
| **Tag** | **Meaning** | **Example** |
| **NN** | Noun (Singular) | "dog", "car" |
| **NNS** | Noun (Plural) | "dogs", "cars" |
| **NNP** | Proper Noun (Singular) | "John", "London" |
| **NNPS** | Proper Noun (Plural) | "Americans", "Indians" |
| **VB** | Verb (Base Form) | "run", "eat" |
| **VBD** | Verb (Past Tense) | "ran", "ate" |
| **VBG** | Verb (Gerund/Progressive) | "running", "eating" |
| **VBN** | Verb (Past Participle) | "eaten", "driven" |
| **VBP** | Verb (Non-3rd person Singular Present) | "run", "eat" |
| **VBZ** | Verb (3rd person Singular Present) | "runs", "eats" |
| **JJ** | Adjective | "big", "beautiful" |
| **JJR** | Comparative Adjective | "bigger", "faster" |
| **JJS** | Superlative Adjective | "biggest", "fastest" |
| **RB** | Adverb | "quickly", "happily" |
| **RBR** | Comparative Adverb | "faster", "sooner" |
| **RBS** | Superlative Adverb | "fastest", "soonest" |
| **PRP** | Personal Pronoun | "he", "she", "it" |
| **PRP$** | Possessive Pronoun | "his", "her", "its" |
| **IN** | Preposition | "in", "on", "at" |
| **DT** | Determiner | "the", "a", "an" |
| **CC** | Coordinating Conjunction | "and", "but", "or" |
| **UH** | Interjection | "wow", "oops" |
| **MD** | Modal Verb | "can", "should", "must" |

These tags are based on **Penn Treebank POS Tags**.

## **2. POS Tagging Approaches**

### **A. Rule-Based POS Tagging**

* Uses **handcrafted grammar rules**.
* Example rule: If a word ends in "-ing", it's likely a **VBG** (gerund).
* **Example Algorithm:** Brill Tagger (transformation-based learning).
* **Limitations:** Fails on unseen words, requires extensive rules.

### **B. Statistical POS Tagging**

* Uses **probabilities** to assign POS tags.
* Example methods:
  + **Hidden Markov Model (HMM)**: Finds the most probable sequence of tags given a sentence.
  + **Conditional Random Fields (CRF)**: Captures dependencies between words.

### **C. Machine Learning-Based POS Tagging**

* **Supervised learning** using labeled datasets (e.g., Penn Treebank).
* Common models:
  + **Naïve Bayes**
  + **Decision Trees**
  + **Neural Networks (BiLSTM, Transformers)**
* **Pros:** Adapts to new words better than rule-based models.
* **Cons:** Needs a lot of labeled data.

## **3. POS Tagging in Python**

**NLTK (Simple but effective)**

import nltk

nltk.download("averaged\_perceptron\_tagger")

text = "The quick brown fox jumps over the lazy dog"

tokens = nltk.word\_tokenize(text)

pos\_tags = nltk.pos\_tag(tokens)

print(pos\_tags)

**Output:**

[('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'), ('fox', 'NN'),

('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]

**SpaCy (Faster, Better for Deep Learning)**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "The quick brown fox jumps over the lazy dog"

doc = nlp(text)

for token in doc:

print(token.text, token.pos\_)

**Output:**

The DET

quick ADJ

brown ADJ

fox NOUN

jumps VERB

over ADP

the DET

lazy ADJ

dog NOUN

**Stanza (Stanford NLP)**

* More **accurate** but **slower** than NLTK.

import stanza

stanza.download("en")

nlp = stanza.Pipeline("en")

doc = nlp("The quick brown fox jumps over the lazy dog.")

for word in doc.sentences[0].words:

print(word.text, word.xpos)

## **4. Advanced POS Tagging Techniques**

**A. BiLSTM + CRF (State-of-the-art POS Tagging)**

* Uses **bidirectional LSTMs** to capture **contextual dependencies**.
* **CRF (Conditional Random Fields)** ensures consistent tagging.
* **Used in:** Deep NLP models like **SpaCy, Stanza, Flair**.

**B. Transformer-Based POS Tagging**

* **BERT, RoBERTa, XLNet**: Use contextual embeddings for tagging.
* **Example:** Fine-tuning BERT for POS tagging.

from transformers import pipeline

pos\_tagger = pipeline("token-classification", model="dslim/bert-base-NER")

text = "The quick brown fox jumps over the lazy dog"

result = pos\_tagger(text)

for word in result:

print(word["word"], word["entity"])

* **Pros:** Context-aware, better for ambiguous words.
* **Cons:** Computationally expensive.

## **5. Why POS Tagging Matters?**

✔ **Parsing:** Helps in dependency parsing for sentence structure analysis.  
✔ **Named Entity Recognition (NER):** Tags nouns for entity extraction.  
✔ **Machine Translation:** Improves grammatical correctness.  
✔ **Sentiment Analysis:** Helps identify adjectives and intensifiers.  
✔ **Speech Recognition:** Assists in disambiguating homophones.

## **6. Common Challenges in POS Tagging**

❌ **Ambiguity**

* "I saw the man with a telescope." (Did I see the man *using* a telescope or *holding* a telescope?)
* "Time flies like an arrow." ("Time" as noun or verb?)

❌ **Unknown Words (OOV - Out of Vocabulary)**

* Rule-based and statistical models struggle with unseen words.
* Transformers solve this with subword tokenization.

❌ **Contextual Variations**

* "Book" (noun) vs. "Book a flight" (verb).
* Older models fail; BERT-based models handle this well.

## **Final Wisdom**

🚀 **Rule-Based POS Tagging is outdated.** Use **BiLSTM + CRF** or **Transformer-based models** for modern NLP.  
🔥 **POS tagging is foundational** for higher-level NLP tasks like Named Entity Recognition (NER), Dependency Parsing, and Sentiment Analysis.  
💡 **If you want speed, use SpaCy. If you want accuracy, use Stanza or Transformer-based models.**

🔑 **POS tagging is not just tagging—it’s the key to understanding language structure.**

# **Named Entity Recognition (NER)**

Named Entity Recognition (NER) is an NLP technique that **identifies and classifies named entities** in text into predefined categories like **persons, organizations, locations, dates, monetary values, etc.**

## **1. Common Named Entity Categories**

|  |  |  |
| --- | --- | --- |
| **Category** | **Description** | **Example** |
| **PERSON** | Names of people | "Elon Musk", "Albert Einstein" |
| **ORG** | Organizations, companies | "Google", "United Nations" |
| **LOC** | Geographic locations | "Mount Everest", "Sahara Desert" |
| **GPE** | Countries, cities, states | "France", "New York" |
| **DATE** | Dates & time expressions | "March 20, 2025", "yesterday" |
| **TIME** | Specific times | "10:30 AM", "midnight" |
| **MONEY** | Monetary values | "$100", "50 Euros" |
| **PERCENT** | Percentages | "75%", "half of the population" |
| **PRODUCT** | Product names | "iPhone 15", "Tesla Model S" |
| **EVENT** | Named events | "World War II", "Olympics 2024" |
| **LAW** | Legal references | "GDPR", "Article 13" |

## **2. Approaches to NER**

### **A. Rule-Based NER**

* Uses **handcrafted rules and regex** (e.g., capitalized words → names, currency symbols → money).
* **Example:**

import re

text = "Elon Musk founded Tesla in 2003."

pattern = r"[A-Z][a-z]+ [A-Z][a-z]+" # Detects names with two capitalized words

print(re.findall(pattern, text))

* # Output: ['Elon Musk', 'Tesla']
* **Limitations:** Fails with unseen patterns, lacks adaptability.

### **B. Statistical & Machine Learning-Based NER**

* **Hidden Markov Models (HMM)** and **Conditional Random Fields (CRF)** predict entity labels using probability.
* Requires **labeled training data**.
* Example: Stanford NER, Spacy, CRF-based models.

### **C. Deep Learning-Based NER**

* Uses **word embeddings** + **LSTMs/BERT** for contextual understanding.
* **BiLSTM + CRF** (State-of-the-art before transformers).
* **Transformer-based models** (BERT, RoBERTa, T5) outperform traditional methods.
* **Example:** spaCy, Hugging Face transformers.

## **3. NER Implementation in Python**

### **A. Using spaCy (Fast & Efficient)**

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = "Elon Musk founded SpaceX in 2002 and Tesla in 2003."

doc = nlp(text)

for ent in doc.ents:

print(ent.text, ent.label\_)

**Output:**

Elon Musk PERSON

SpaceX ORG

2002 DATE

Tesla ORG

2003 DATE

* **Pros:** Fast, optimized for production.
* **Cons:** Less accurate than deep-learning models.

### **B. Using Hugging Face Transformers (State-of-the-Art)**

from transformers import pipeline

ner = pipeline("ner", model="dbmdz/bert-large-cased-finetuned-conll03-english")

text = "Barack Obama was the 44th president of the United States."

results = ner(text)

for entity in results:

print(entity["word"], entity["entity"])

**Output:**

Barack B-PER

Obama I-PER

United B-LOC

States I-LOC

* **Pros:** **Highly accurate**, context-aware (e.g., "Apple" as fruit vs. company).
* **Cons:** **Slower** and requires **GPU** for large-scale inference.

### **C. Using Stanford NER (CRF-Based)**

from nltk.tag import StanfordNERTagger

from nltk.tokenize import word\_tokenize

st = StanfordNERTagger('english.all.3class.distsim.crf.ser.gz')

text = "Google was founded by Larry Page and Sergey Brin in 1998."

words = word\_tokenize(text)

print(st.tag(words))

**Output:**

[('Google', 'ORGANIZATION'), ('Larry', 'PERSON'), ('Page', 'PERSON'), ('1998', 'DATE')]

* **Pros:** Strong rule-based + ML model.
* **Cons:** Requires Java setup, slower than spaCy.

## **4. Advanced NER Techniques**

### **A. BiLSTM + CRF (Deep Learning + Structured Output)**

* **Why?** LSTMs learn **long-range dependencies**, and CRFs enforce **sequential constraints**.
* **Pipeline:**
  1. Convert text → Word Embeddings (Word2Vec, GloVe, FastText).
  2. Feed embeddings to a **Bidirectional LSTM**.
  3. Pass LSTM outputs to a **CRF layer** for structured prediction.

### **B. Transformer-Based NER (State-of-the-Art)**

* **BERT, RoBERTa, T5, GPT** outperform traditional ML models.
* **Why?** Transformers understand context, resolving ambiguities (e.g., "Amazon" as a river vs. a company).
* **Fine-tuning BERT for NER:**

from transformers import AutoModelForTokenClassification, AutoTokenizer

import torch

model\_name = "dbmdz/bert-large-cased-finetuned-conll03-english"

model = AutoModelForTokenClassification.from\_pretrained(model\_name)

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

text = "Microsoft acquired LinkedIn for $26 billion in 2016."

tokens = tokenizer(text, return\_tensors="pt")

output = model(\*\*tokens)

print(output)

* **Pros:** Best accuracy, context-aware.
* **Cons:** Computationally expensive.

## **5. Challenges in NER**

|  |  |  |
| --- | --- | --- |
| **Challenge** | **Explanation** | **Example** |
| **Ambiguity** | Same word, different meanings. | "Apple" (fruit) vs. "Apple Inc." (company) |
| **Complex Entity Names** | Multi-word names may be hard to detect. | "Bank of America", "New York Times" |
| **Entity Overlap** | Nested entities within other entities. | "New York University" (ORG) inside "New York" (GPE) |
| **Misspellings & Variations** | Entities appear in different forms. | "IBM" vs. "International Business Machines" |
| **Low-Resource Languages** | Lack of labeled training data for some languages. | Many African or Indigenous languages |

## **6. When to Use Which NER Model?**

|  |  |
| --- | --- |
| **Use Case** | **Recommended Model** |
| **Fast & Lightweight** | spaCy |
| **High Accuracy** | BERT-based models (Hugging Face) |
| **Domain-Specific (e.g., Medical, Legal)** | Fine-tuned BERT (SciBERT, BioBERT, LegalBERT) |
| **Production-Ready ML Model** | BiLSTM + CRF |
| **Offline/Java-Based** | Stanford NER |

## **7. Applications of NER**

✔ **Chatbots & Virtual Assistants** – Extracting user intents ("Book a flight to Paris").  
✔ **Search Engines** – Identifying entities for better search results.  
✔ **Finance & Business** – Recognizing company names in news.  
✔ **Healthcare & Biomedical NLP** – Identifying diseases, drugs, and genes.  
✔ **Legal Text Analysis** – Extracting references to laws and regulations.

## **Final Wisdom**

🚀 **Rule-Based NER is obsolete. Use ML/DL-based models.**  
🔥 **For speed, use spaCy. For accuracy, use BERT or BiLSTM+CRF.**  
💡 **NER is crucial for search engines, chatbots, and intelligent document processing.**  
🔑 **Fine-tune transformer models for domain-specific NER (Medical, Legal, Finance).**

# **Text Representation (Feature Engineering)**

Text representation is the process of converting raw text into a format that can be understood and processed by machine learning and deep learning models.

Representing text as a vector is also known as **Vector Space Model**.

Raw Text 🡪 Number Vector 🡪 Machine Learning

### **1. Why Do We Need Text Representation?**

* Machine learning models work with **numerical data**, but text is unstructured.
* Converting text into numerical form enables models to **analyze, classify, and generate** text.
* Good representation preserves the **semantic meaning** of words and sentences.

## **2. Types of Text Representation**

### **A. Basic Text Representations (Classical Methods)**

1. **One-Hot Encoding (OHE)**
   * Each word is represented as a **binary vector** where one element is 1 (word presence), and all others are 0.
   * **Example:**
   * Vocabulary: ["apple", "banana", "orange", "grape"]
   * "apple” → [1, 0, 0, 0]
   * "banana" → [0, 1, 0, 0]
   * **Limitations:**
     + **Sparsity** (high-dimensional for large vocabularies).
     + **No semantic meaning** (e.g., "king" and "queen" are unrelated).
2. **Bag of Words (BoW)**
   * Represents text as **word frequencies** in a fixed vocabulary.
   * Ignores word order and context.

**Example:**

Text 1: "I love NLP and deep learning"

Text 2: "Deep learning is amazing"

|  |  |  |
| --- | --- | --- |
| **Word** | **Text 1** | **Text 2** |
| I | 1 | 0 |
| love | 1 | 0 |
| NLP | 1 | 0 |
| and | 1 | 0 |
| deep | 1 | 1 |
| learning | 1 | 1 |
| is | 0 | 1 |
| amazing | 0 | 1 |

* + **Limitations:**
    - **High dimensionality** for large vocabularies.
    - **No context awareness** (word order is ignored).

1. **Term Frequency - Inverse Document Frequency (TF-IDF)**
   * Weighs words based on their importance across multiple documents.
   * Formula: TF-IDF=TF×IDF
   * where:
     + **TF (Term Frequency)** = Frequency of the word in a document.
     + **IDF (Inverse Document Frequency)** = How rare the word is across documents.
   * **Example:**
   * Document 1: "NLP is great for text processing"
   * Document 2: "Text processing is fun"
   * Common words like **"is"** get lower importance, while rare words like **"NLP"** get higher importance.
   * **Limitations:**
     + Still **ignores word order** and **context**.

## **B. Dense Word Embeddings (Advanced Methods)**

Word embeddings **map words to dense vectors** that capture their semantic meaning.

**1. Word2Vec (Mikolov et al., 2013)**

* Uses **neural networks** to learn word relationships.
* Two models:
  + **CBOW (Continuous Bag of Words):** Predicts a target word from its surrounding context.
  + **Skip-Gram:** Predicts surrounding words given a target word.
* Example (Word2Vec representation):
* king → [0.21, -0.35, 0.48, ...]
* queen → [0.20, -0.37, 0.50, ...]
* **Pros:**
  + Captures **semantic similarity** (e.g., "king" and "queen" are close in vector space).
  + Efficient training on large text corpora.
* **Cons:**
  + Doesn't handle **polysemy** (e.g., "bank" as a financial institution vs. riverbank).

**2. GloVe (Pennington et al., 2014)**

* Learns word representations based on **global word co-occurrence** statistics.
* Captures both **local context** (like Word2Vec) and **global corpus statistics**.
* **Example:**
* Paris - France + Italy ≈ Rome
* **Pros:**
  + Retains **word frequency information**.
  + Outperforms Word2Vec in many cases.

**3. FastText (Bojanowski et al., 2017)**

* Improves Word2Vec by using **subword information** (character n-grams).
* Example (word split into character n-grams):
* "apple" → ["<ap", "app", "ppl", "ple", "le>"]
* **Pros:**
  + Handles **out-of-vocabulary (OOV)** words better than Word2Vec/GloVe.
  + Captures **morphological** structure (useful for languages like German or Turkish).

## **C. Contextual Word Embeddings (State-of-the-Art)**

These methods generate **dynamic embeddings** based on **sentence context**.

**1. ELMo (Embeddings from Language Models, 2018)**

* Uses **BiLSTMs** to create word embeddings that vary based on context.
* Example:
* "bank" in "I deposited money in the bank" ≠ "I sat by the river bank"
* **Pros:**
  + Handles **polysemy** (same word, different meanings).

**2. BERT (Bidirectional Encoder Representations from Transformers, 2018)**

* Transformer-based model that learns **deep contextualized** embeddings.
* **Pre-trained on massive datasets**, then fine-tuned for specific tasks.
* Example (BERT embeddings for "apple" in different contexts):
* "Apple Inc. released a new iPhone." → [Vector A]
* "I ate a fresh apple." → [Vector B]
* **Pros:**
  + Best-in-class contextual understanding.
  + Handles long-range dependencies in text.
* **Cons:**
  + Computationally expensive.

**3. GPT (Generative Pre-trained Transformer)**

* Like BERT but optimized for **generative tasks** (e.g., text completion).
* Used in **ChatGPT**, text summarization, and chatbot applications.

## 3. Comparing Text Representation Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Sparsity** | **Captures Meaning?** | **Handles OOV Words?** | **Context-Aware?** |
| **One-Hot Encoding** | High | ❌ No | ❌ No | ❌ No |
| **Bag of Words** | High | ❌ No | ❌ No | ❌ No |
| **TF-IDF** | High | ❌ No | ❌ No | ❌ No |
| **Word2Vec** | Low | ✅ Yes | ❌ No | ❌ No |
| **GloVe** | Low | ✅ Yes | ❌ No | ❌ No |
| **FastText** | Low | ✅ Yes | ✅ Yes | ❌ No |
| **ELMo** | Low | ✅ Yes | ✅ Yes | ✅ Yes |
| **BERT** | Low | ✅ Yes | ✅ Yes | ✅ Yes |

## **4. Choosing the Right Text Representation**

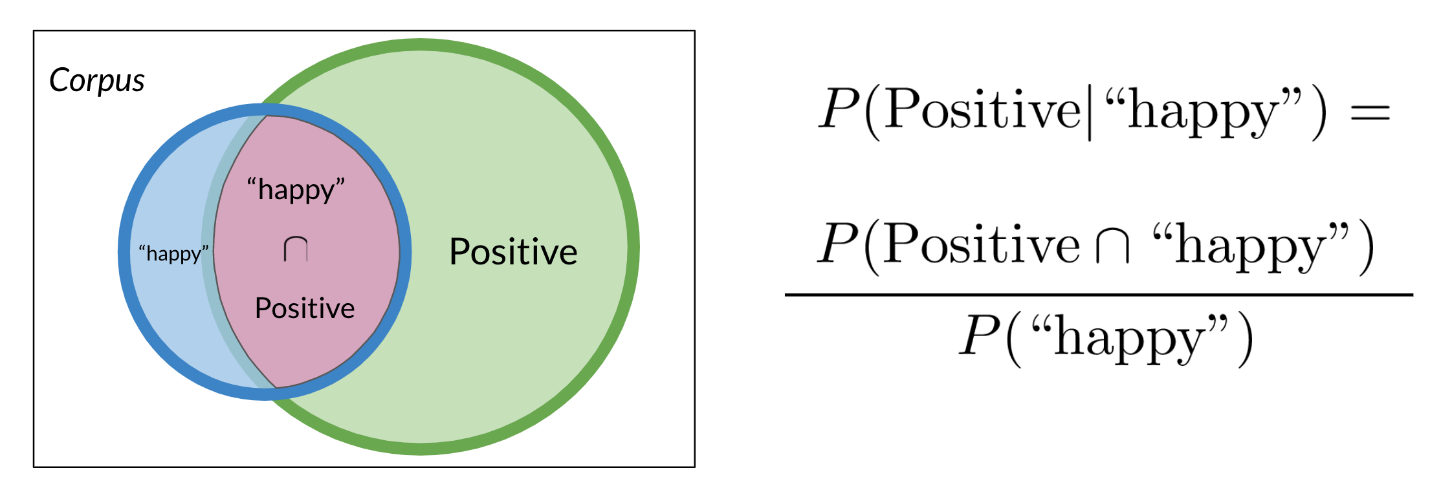
* **Basic NLP tasks (spam detection, sentiment analysis)** → **TF-IDF or Word2Vec**.
* **Large-scale NLP applications (search engines, chatbots)** → **BERT or GPT**.
* **Domain-specific NLP (biomedical, legal, finance)** → **Fine-tuned BERT models (BioBERT, LegalBERT)**.
* **Low-resource languages** → **FastText (handles unseen words well)**.

## **Final Wisdom**

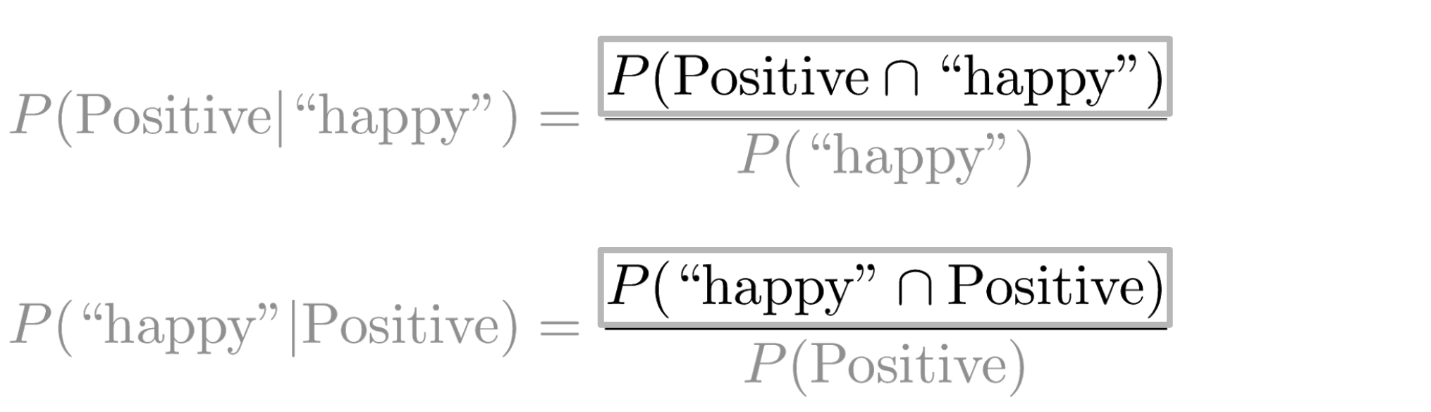
🚀 **Classical methods (BoW, TF-IDF) are simple but outdated.**  
🔥 **Word2Vec, GloVe, and FastText improve semantic understanding.**  
💡 **Contextual embeddings (ELMo, BERT, GPT) are the future of NLP.**  
🔑 **Choose representations based on task complexity and computational constraints.**

# **Bayes Rule**

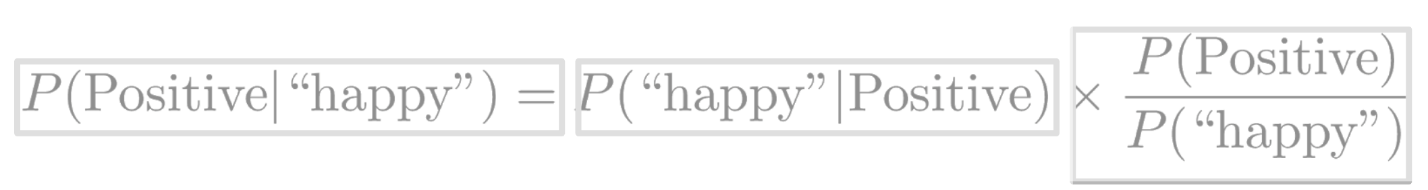
Conditional probabilities help us reduce the sample search space. For example, given a specific event already happened, i.e. we know the word is happy:



Then you would only search in the blue circle above. The numerator will be the red part and the denominator will be the blue part. This leads us to conclude the following:



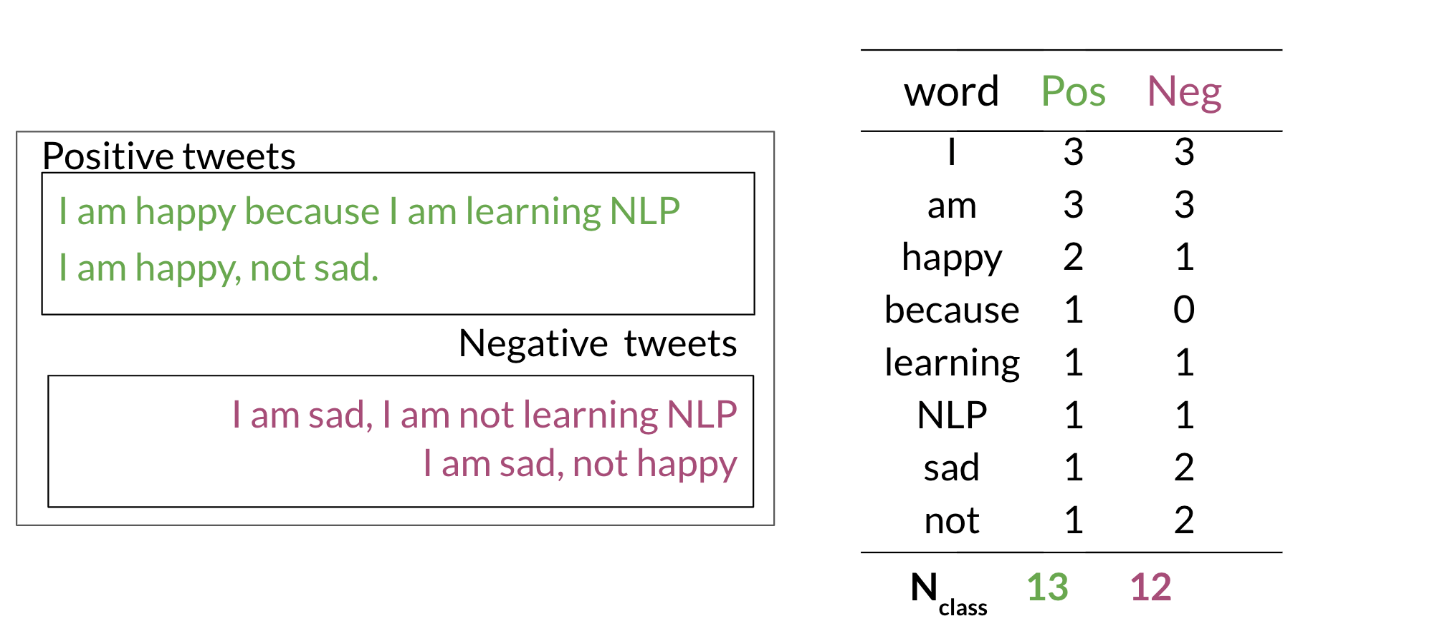
Substituting the numerator in the right-hand side of the first equation, you get the following:



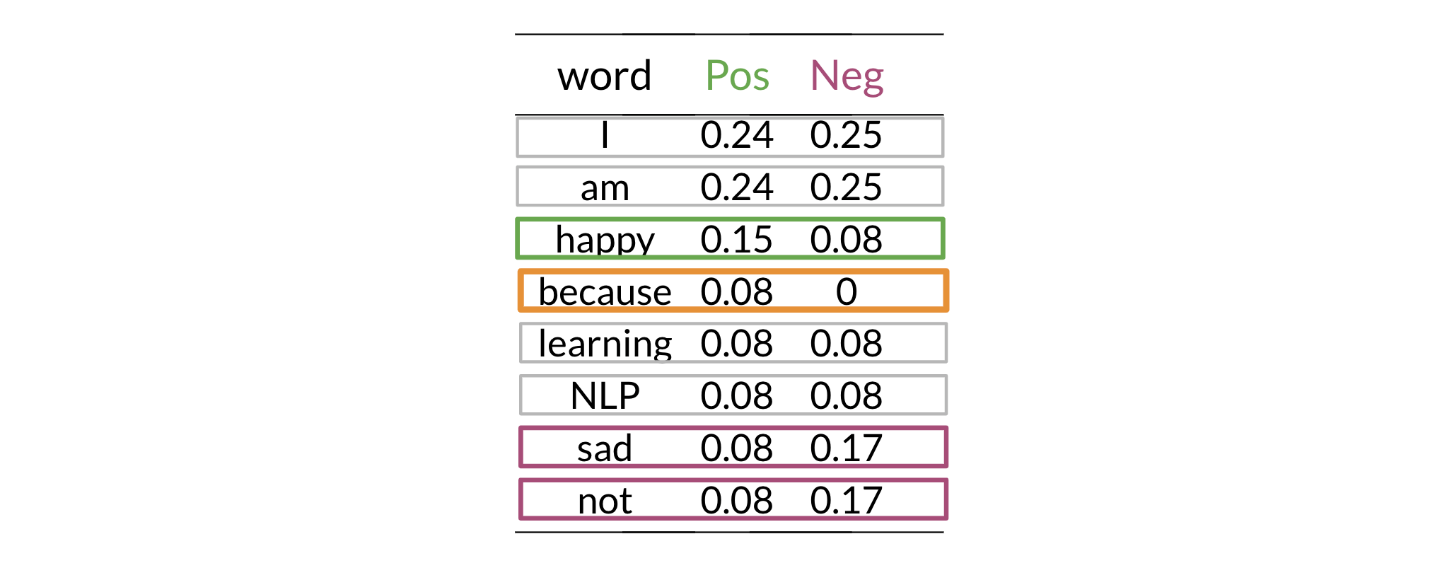
Note that we multiplied by P(positive) to make sure we don't change anything. That concludes Bayes Rule which is defined as

**P (X ∣ Y) =**

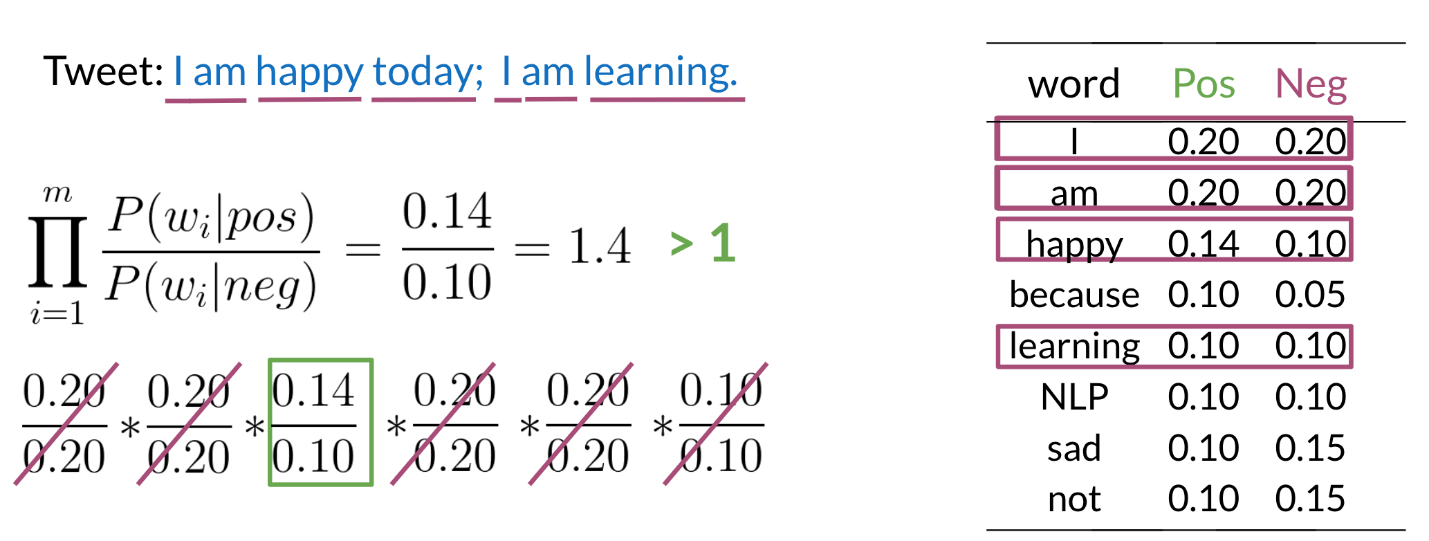
To build a classifier, we will first start by creating conditional probabilities given the following table:



This allows us compute the following table of probabilities:



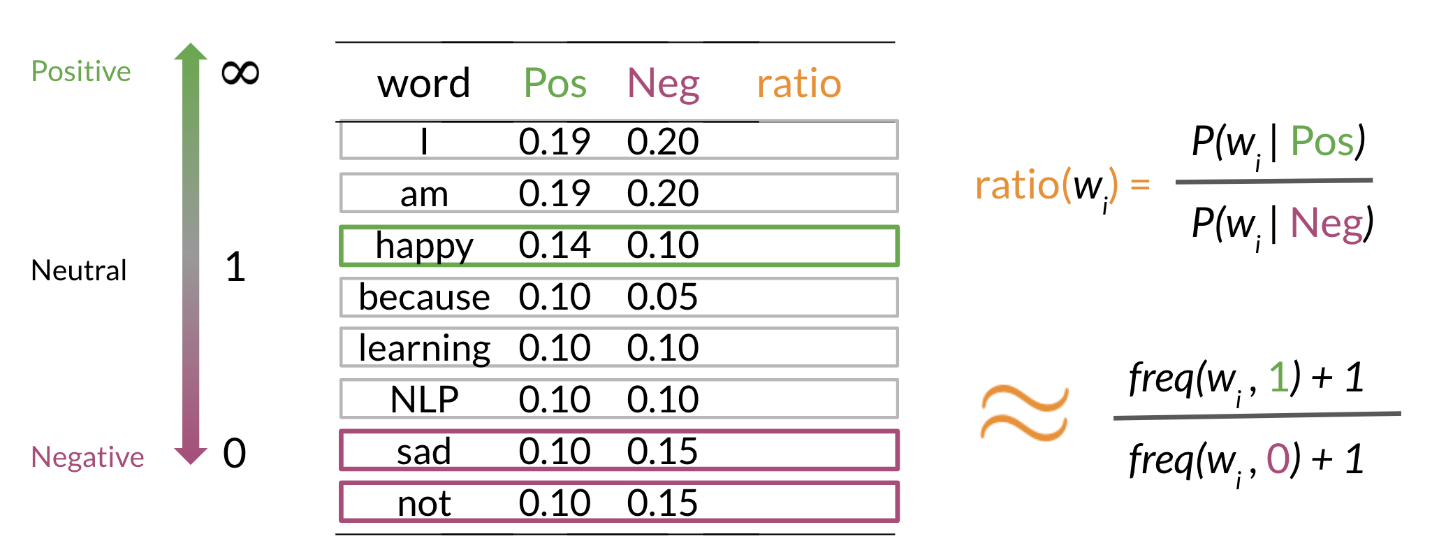
Once you have the probabilities, you can compute the likelihood score as follows



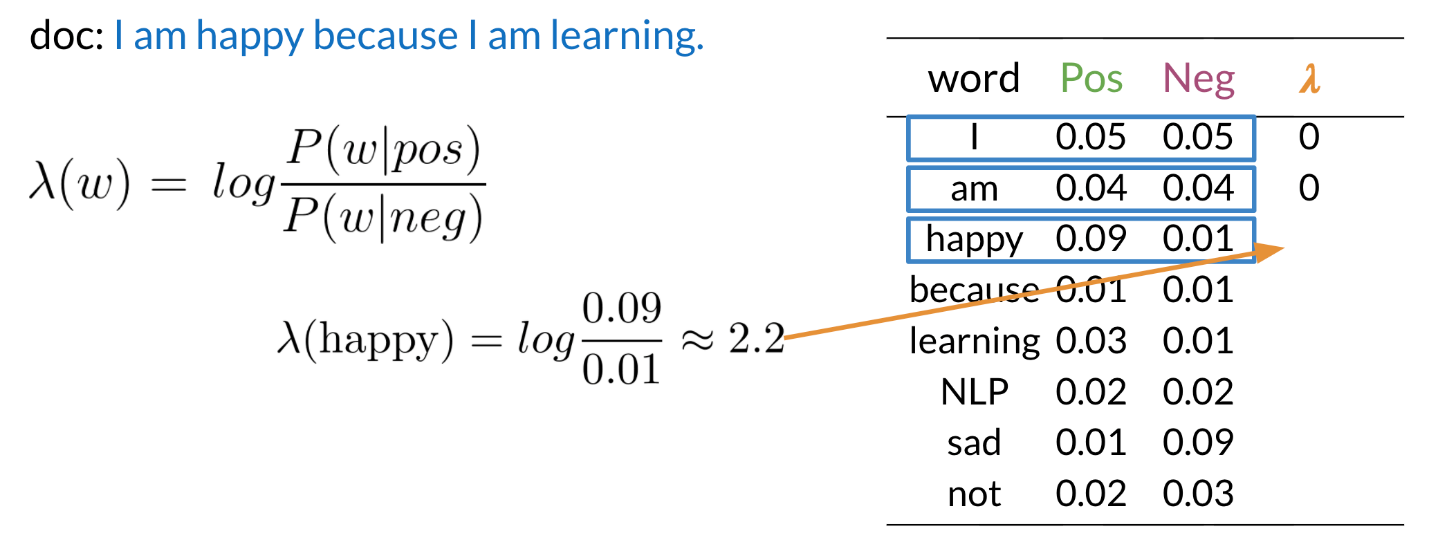
A score greater than 1 indicates that the class is positive, otherwise it is negative.

## **Log Likelihood**

To compute the log likelihood, we need to get the ratios and use them to compute a score that will allow us to decide whether a tweet is positive or negative. The higher the ratio, the more positive the word is:

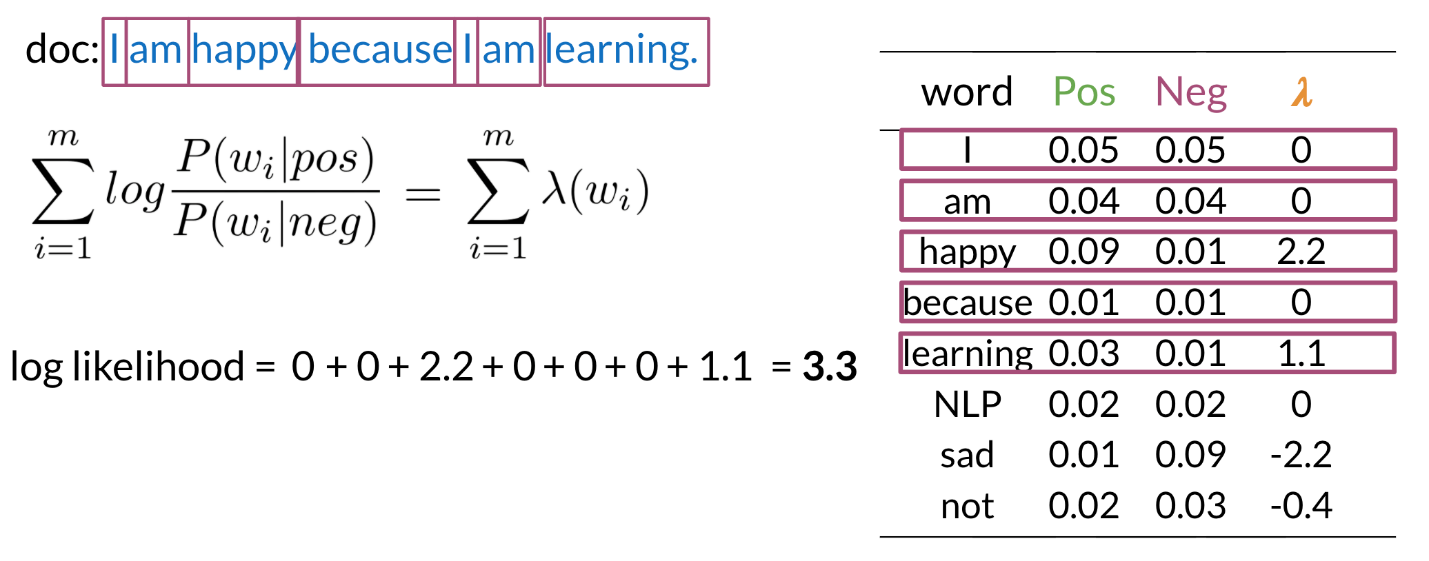


The first component is called the log prior and the second component is the log likelihood. We further introduce λlambda as follows:



Having the λlambda dictionary will help a lot when doing inference.

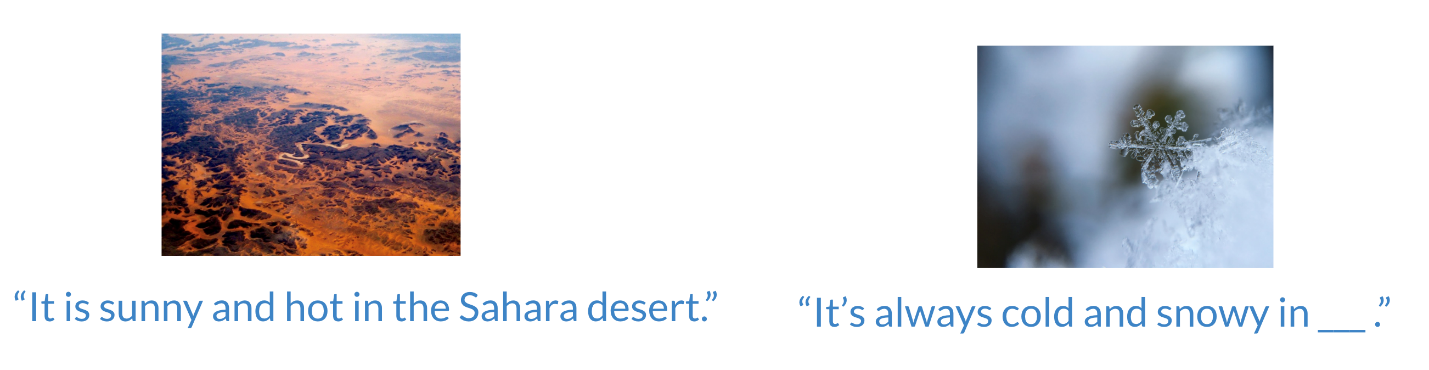
Once you computed the λlambda dictionary, it becomes straightforward to do inference:



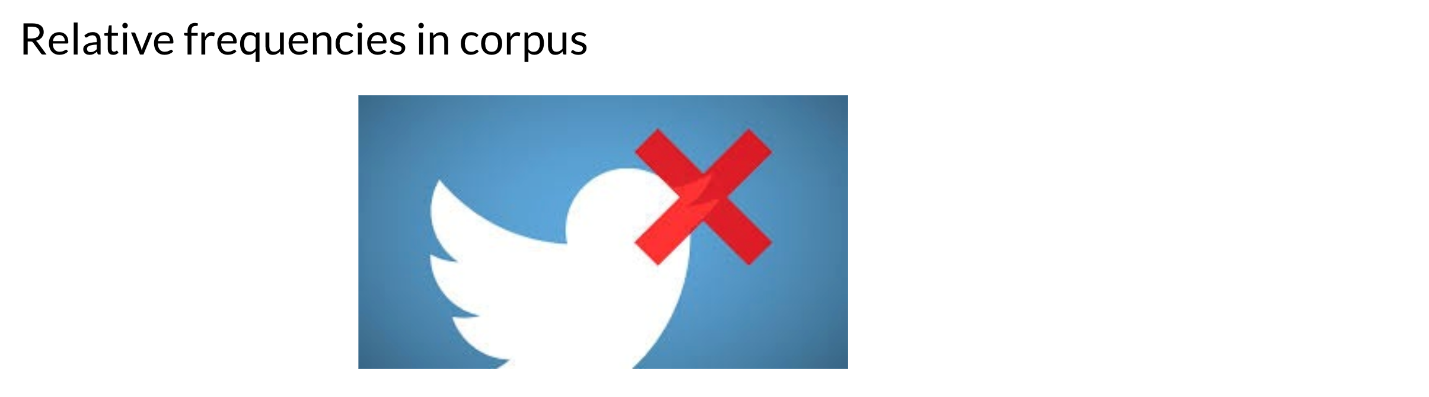
As you can see above, since 3.3 > 0, point, 3, is greater than, 0, we will classify the document to be positive. If we got a negative number we would have classified it to the negative class.

## **Naïve Bayes Assumptions**

Naïve Bayes makes the independence assumption and is affected by the word frequencies in the corpus. For example, if you had the following



In the first image, you can see the word sunny and hot tend to depend on each other and are correlated to a certain extent with the word "desert". Naive Bayes assumes independence throughout. Furthermore, if you were to fill in the sentence on the right, this naive model will assign equal weight to the words "spring, summer, fall, winter".

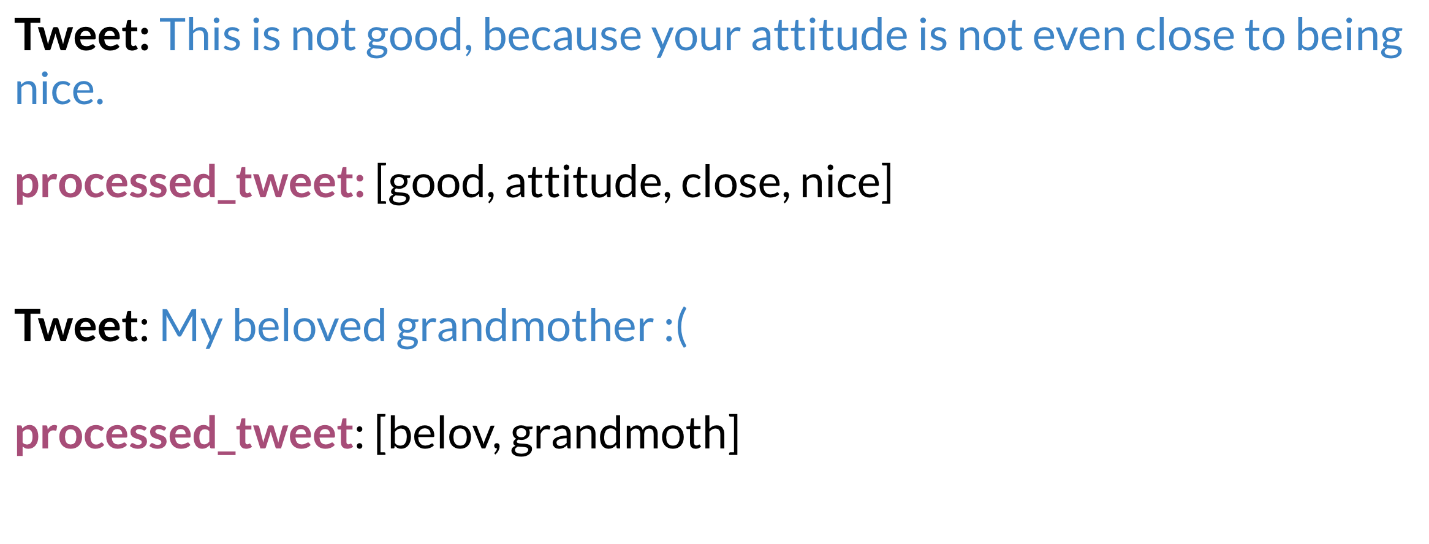


On Twitter, there are usually more positive tweets than negative ones. However, some "clean" datasets you may find are artificially balanced to have to the same amount of positive and negative tweets. Just keep in mind, that in the real world, the data could be much noisier.

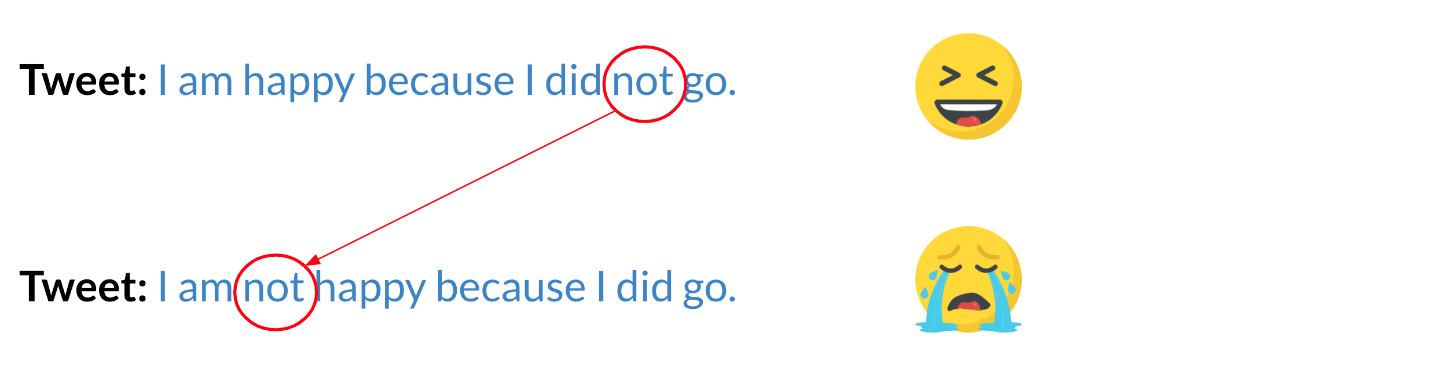
## **Error Analysis**

There are several mistakes that could cause you to misclassify an example or a tweet. For example,

* Removing punctuation
* Removing words



* Word order

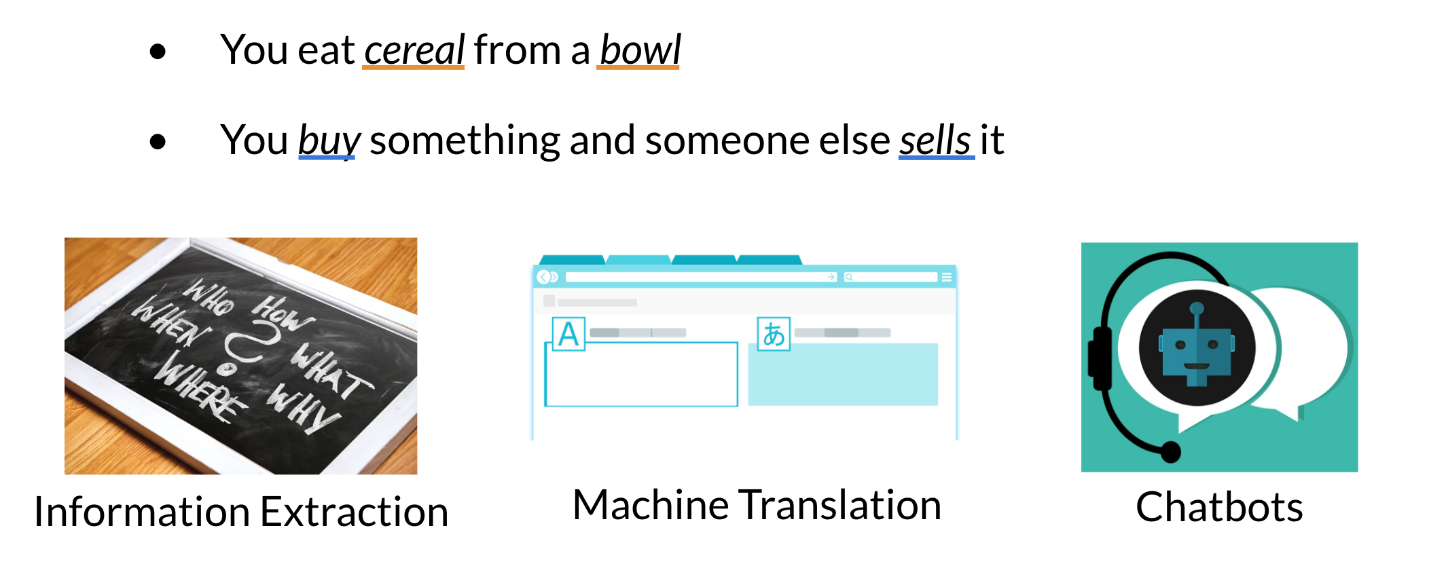


* Adversarial attacks

These include sarcasm, irony, euphemisms.

# **Vector Space Model**

Vector spaces are fundamental in many applications in NLP. If you were to represent a word, document, tweet, or any form of text, you will probably be encoding it as a vector. These vectors are important in tasks like information extraction, machine translation, and chatbots. Vector spaces could also be used to help you identify relationships between words as follows:

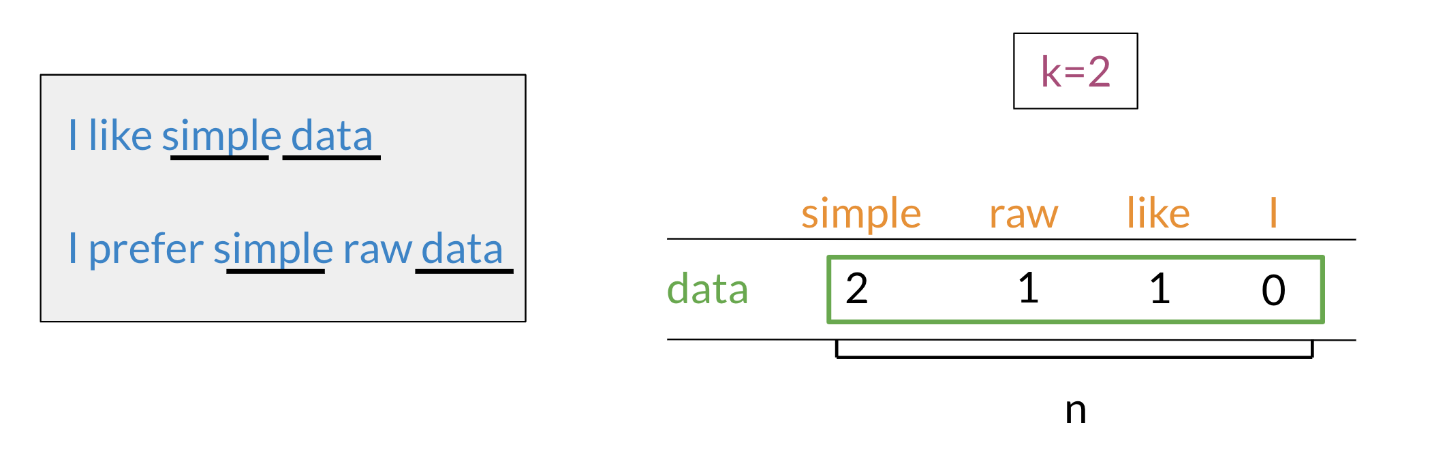


The famous quote by Firth says, **"You shall know a word by the company it keeps".** When learning these vectors, you usually make use of the neighboring words to extract meaning and information about the center word. If you were to cluster these vectors together, as you will see later in this specialization, you will see those adjectives, nouns, verbs, etc. tend to be near one another. Another cool fact, is that synonyms and antonyms are also very close to one another. This is because you can easily interchange them in a sentence and they tend to have similar neighboring words!

## **Word by Word and Word by Doc**

**Word by Word Design**

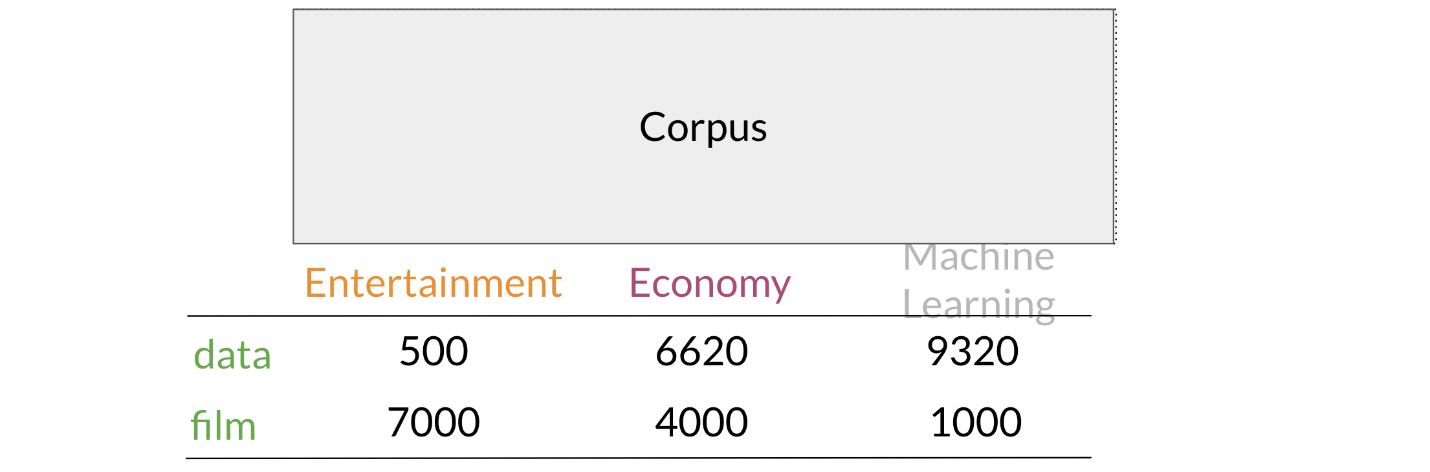
We will start by exploring the word-by-word design. Assume that you are trying to come up with a vector that will represent a certain word. One possible design would be to create a matrix where each row and column corresponds to a word in your vocabulary. Then you can iterate over a document and see the number of times each word shows up next each other word. You can keep track of the number in the matrix. In the video I spoke about a parameter K. You can think of K*K*K as the bandwidth that decides whether two words are next to each other or not.



In the example above, you can see how we are keeping track of the number of times words occur together within a certain distance k. At the end, you can represent the word data, as a vector v= [2,1,1,0].

**Word by Document Design**

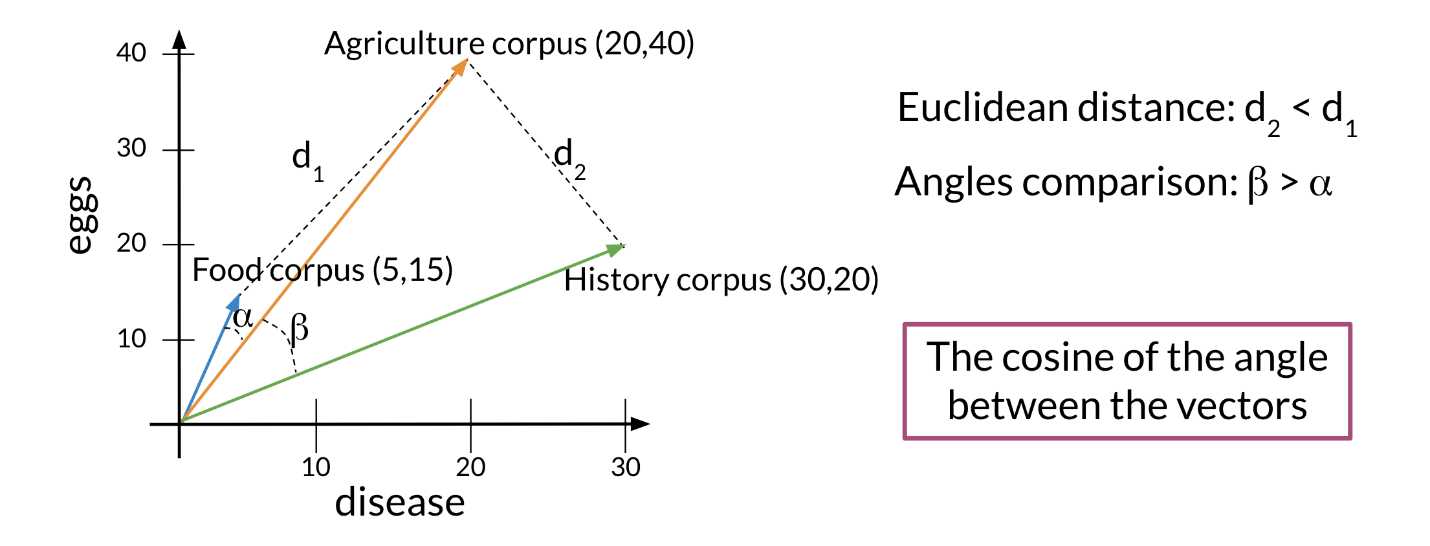
You can now apply the same concept and map words to documents. The rows could correspond to words and the columns to documents. The numbers in the matrix correspond to the number of times each word showed up in the document.



You can represent the entertainment category, as a vector v= [500,7000]. You can then also compare categories as follows by doing a simple plot.

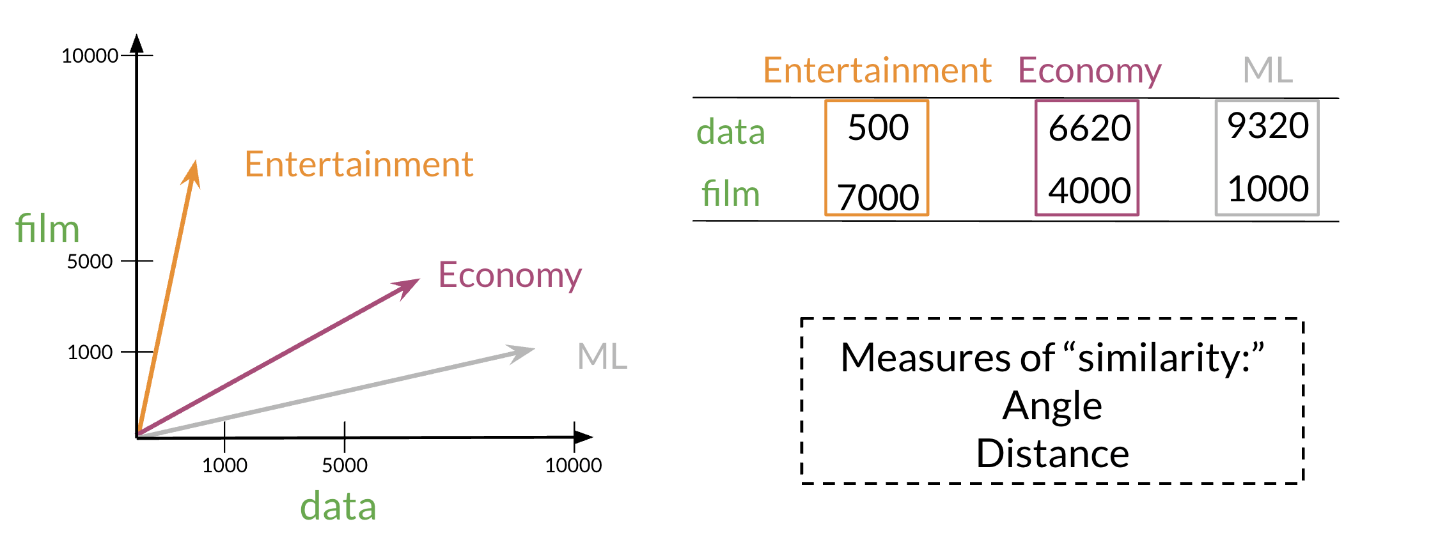
## **Cosine Similarity: Intuition**

One of the issues with Euclidean distance is that it is not always accurate and sometimes we are not looking for that type of similarity metric. For example, when comparing large documents to smaller ones with Euclidean distance one could get an inaccurate result. Look at the diagram below:



Normally the **food** corpus and the **agriculture** corpus are more similar because they have the same proportion of words. However, the food corpus is much smaller than the agriculture corpus. To further clarify, although the history corpus and the agriculture corpus are different, they have a smaller Euclidean distance. Hence d2<d1, start subscript, 2, end subscript, is less than, d, start subscript, 1, end subscript.

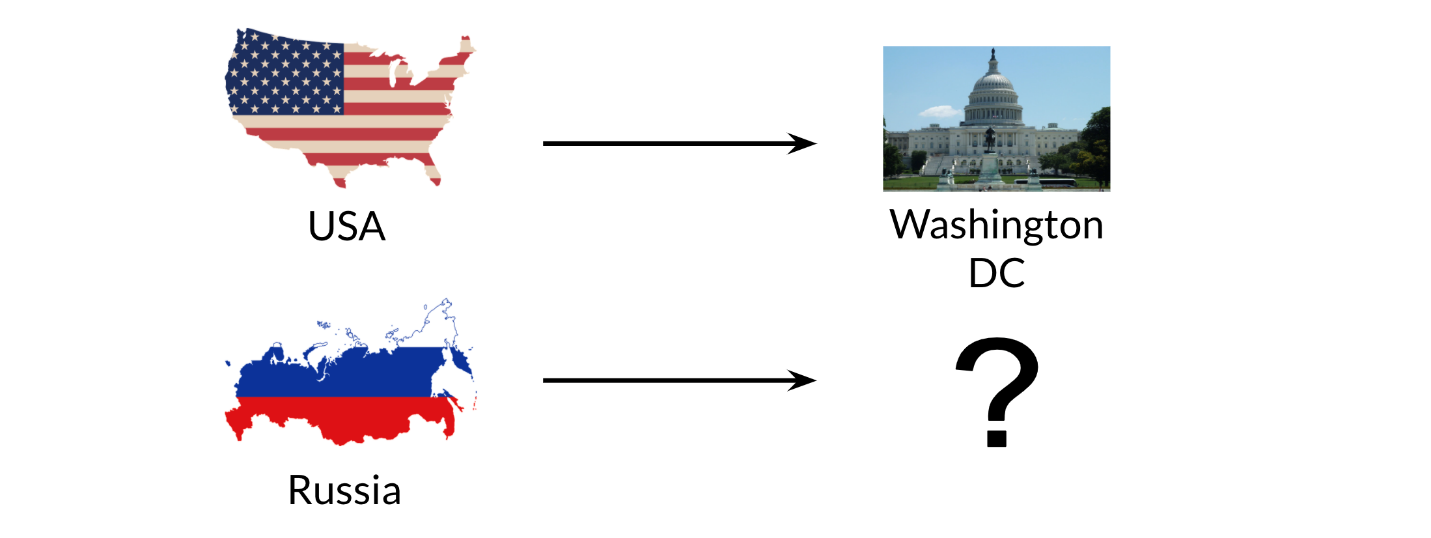
To solve this problem, we look at the cosine between the vectors. This allows us to compare B and α.



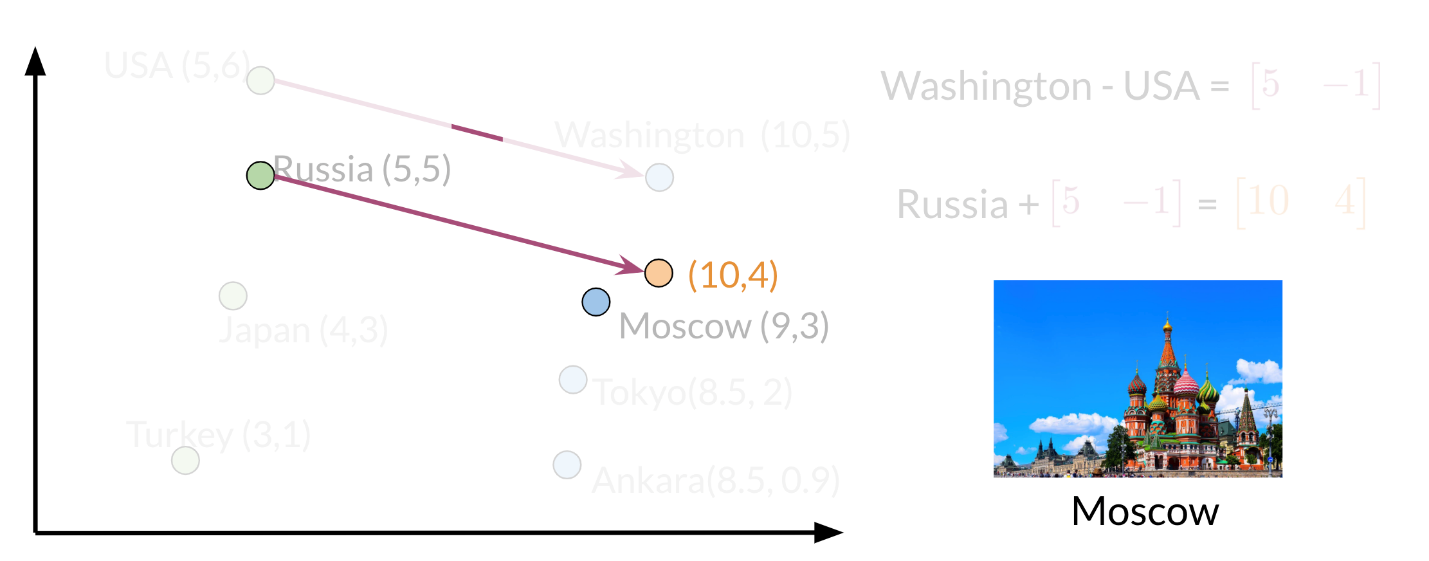
You will see how you can use the angle between two vectors to measure similarity.

### **Manipulating Words in Vector Spaces**

You can use word vectors to actually extract patterns and identify certain structures in your text. For example:



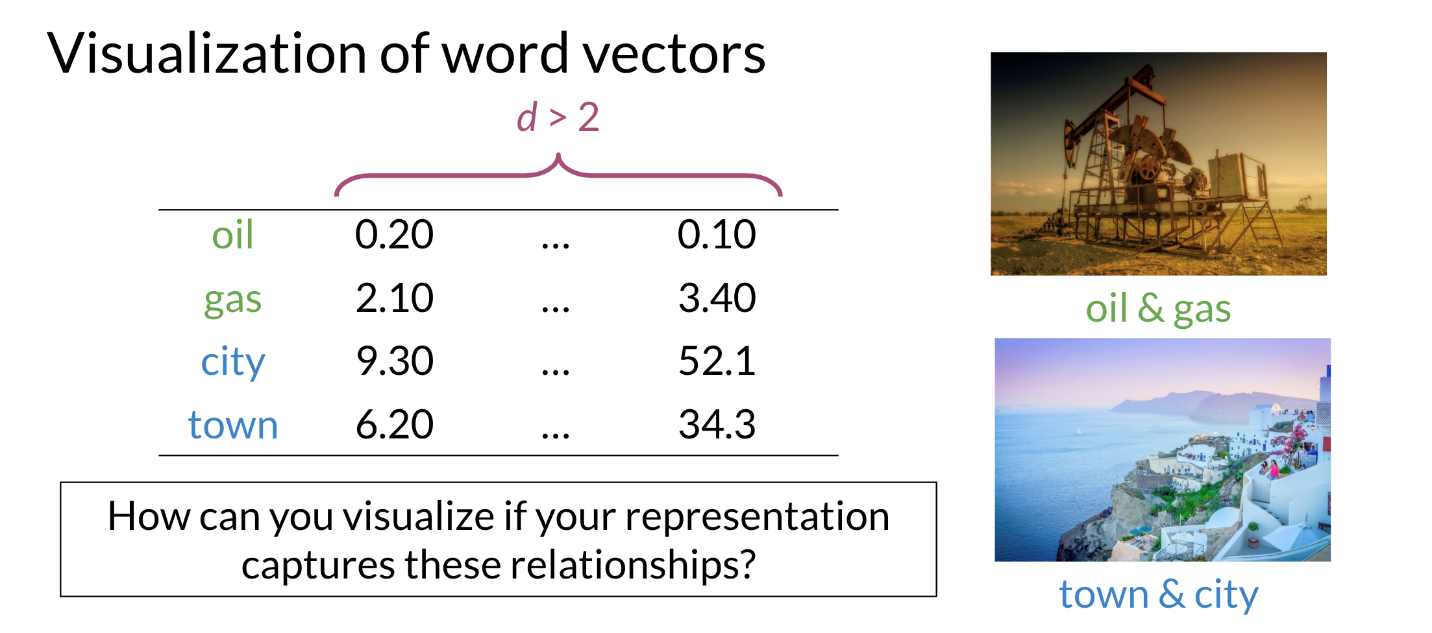
You can use the word vector for Russia, USA, and DC to actually compute a **vector** that would be very similar to that of Moscow. You can then use cosine similarity of the **vector** with all the other word vectors you have and you can see that the vector of Moscow is the closest. Isn't that cool?



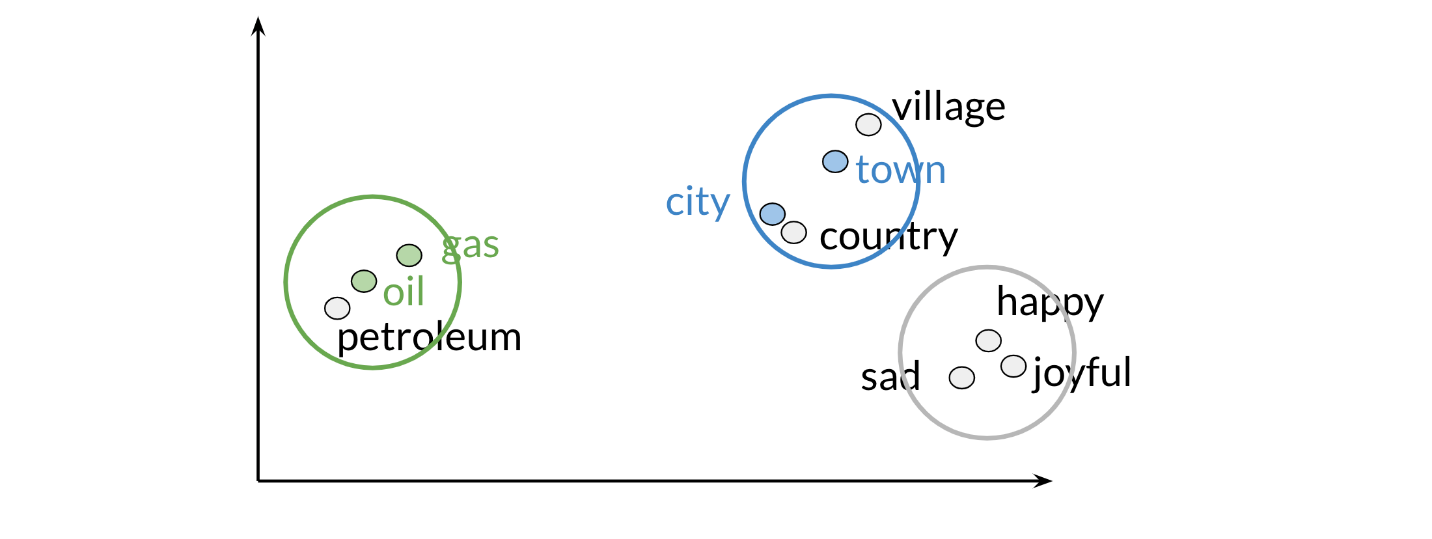
Note that the distance (and direction) between a country and its capital is relatively the same. Hence manipulating word vectors allows you identify patterns in the text.

## **Visualization and PCA**

Principal component analysis is an unsupervised learning algorithm which can be used to reduce the dimension of your data. As a result, it allows you to visualize your data. It tries to combine variances across features. Here is a concrete example of PCA:



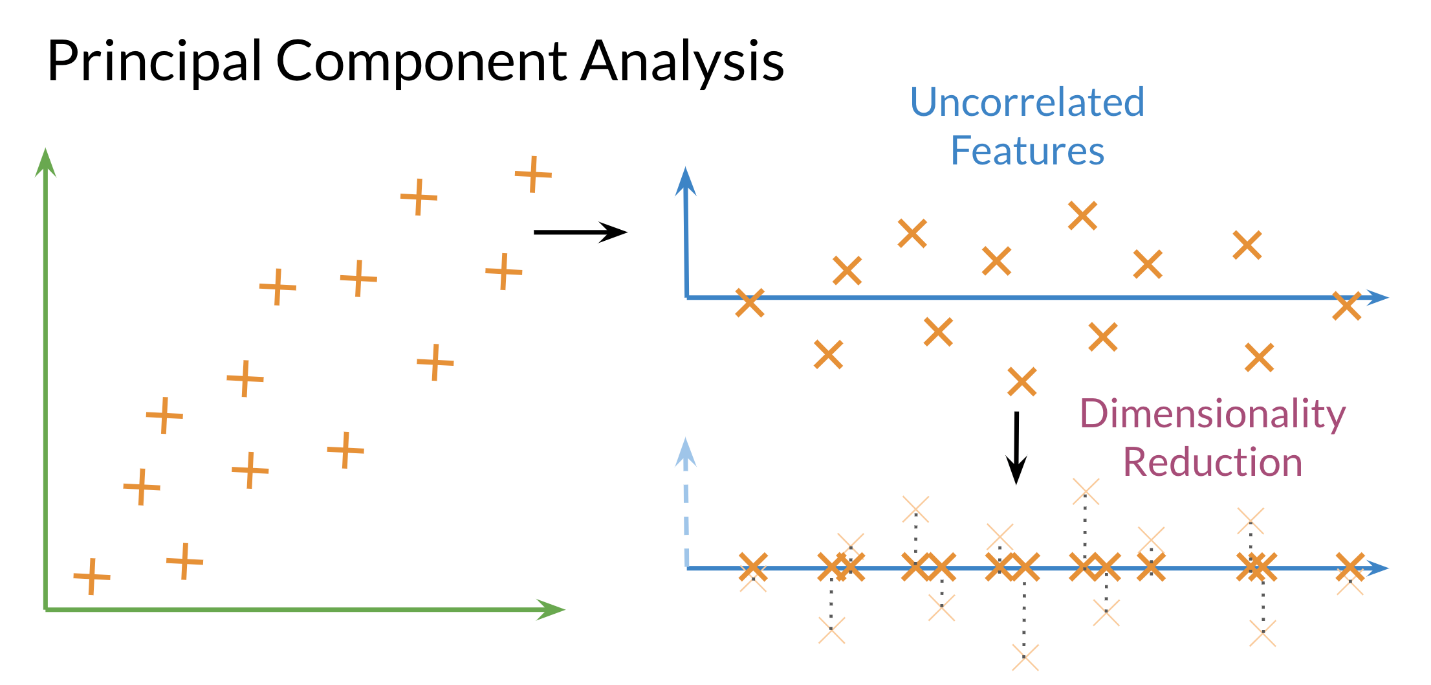
Note that when doing PCA on this data, you will see that oil & gas are close to one another and town & city are also close to one another. To plot the data you can use PCA to go from d>2, is greater than, 2 dimensions to d=2, equals, 2.



Those are the results of plotting a couple of vectors in two dimensions. Note that words with similar part of speech (POS) tags are next to one another. This is because many of the training algorithms learn words by identifying the neighboring words. Thus, words with similar POS tags tend to be found in similar locations.

# **PCA Algorithm**

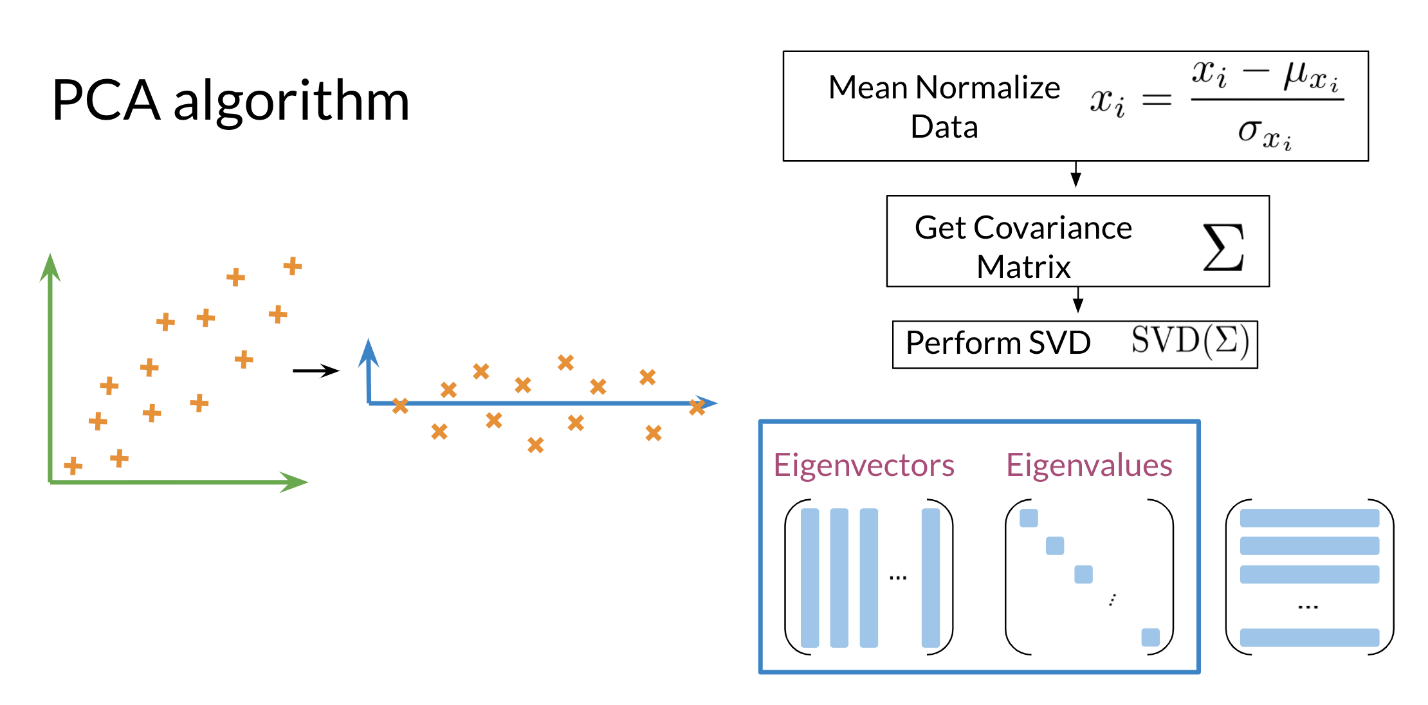
PCA is commonly used to reduce the dimension of your data. Intuitively the model collapses the data across principal components. You can think of the first principal component (in a 2D dataset) as the line where there is the most amount of variance. You can then collapse the data points on that line. Hence you went from 2D to 1D. You can generalize this intuition to several dimensions.



**Eigenvector**: the resulting vectors, also known as the uncorrelated features of your data

**Eigenvalue:** the amount of information retained by each new feature. You can think of it as the variance in the eigenvector.

Also, each **eigenvalue** has a corresponding **eigenvector**. The eigenvalue tells you how much variance there is in the eigenvector. Here are the steps required to compute PCA:



**Steps to Compute PCA:**

* Mean normalize your data
* Compute the covariance matrix
* Compute Singular Value Decomposition on your covariance matrix. This returns [USV]=svd(Σ). The three matrices U, S, V are drawn above. U is labelled with eigenvectors, and S is labelled with eigenvalues.
* You can then use the first n columns of vector U, to get your new data by multiplying XU [:, 0: n]*.*